COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

R1B1 Detailed Design

Contract DBM-9713-NMS
TSR # 9901961
Document # M361-DS-002R0

January 21, 2000
By
Computer Sciences Corporation and PB Farradyne Inc

Table of Contents

1 Introduction

1.1 Purpose

This document describes the detailed design of the Chart II system software for Release 1, Build 1. This design is driven by the Release 1, Build 1 requirements as stated in document M361-RS-001-00, "***CHART II System Requirements Specification For Release 1 Build 1*** " and further refines the high level design presented in document M361-DS-001, ***"R1B1 High Level Design".***

1.2 Objectives

The main objective of this design is to provide software developers with details regarding the implementation of the service applications used to satisfy the requirements of Release 1, Build 1 of the Chart II system.

This design also serves to provide documentation to those outside of the software development community to show how the requirements are being accounted for in the software design.

1.3 Scope

This design is limited to Release 1, Build 1 of the Chart II system and the requirements as stated in the aforementioned requirements document. Additionally, this design document includes only the design of CHART II services and does not include the design of the Graphical User Interface.

1.4 Acronyms

The following acronyms appear throughout this document:

BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DMS	Dynamic Message Sign
FMS	Field Management Station
GUI	Graphical User Interface
IDL	Interface Definition Language
OMG	Object Management Group

ORB	Object Request Broker
R1B1	Release 1, Build 1 of the CHART II System
UML	Unified Modeling Language

1.5 References

CHART II System Requirements Specification For Release 1 Build 1, document number M361-RS-001-00, Computer Sciences Corporation and PB Farradyne, Inc.

R1B1 High Level Design, document number M361-DS-001-00, Computer Sciences Corporation and PB Farradyne, Inc.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, OMG Document 98-02-33

Martin Fowler and Kendall Scott, *UML Distilled,* Addison-Wesley, 1997

1.6 Design Process

As in the high level design, object oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), a de facto standard for diagramming object oriented designs.

In the high level design, system interfaces were identified and specified. These interfaces were partitioned into logical groupings of packages. This design serves to fill in the details necessary to implement each of the system interfaces identified in the high level design.

In this design, each package identified in the high level design is addressed separately with its own class diagram and sequence diagrams for major operations included in the package's interfaces. Additionally, packages needed for implementation but not present in the high level design are included in this design, with each of these also having its own class diagram and sequence diagrams. Packages are also included for third party software that is needed by the CHART II software, such as the ORB and Java classes. Only classes and methods shown on the sequence diagrams are included in diagrams for third party products.

The design process for each package involved starting with a class diagram including interfaces from the high level design, and filling in details to the class diagram to move toward implementation. Sequence diagrams were then used to show how the functionality is to be carried out. An iterative process was used to enhance the class diagram as sequence diagrams identified missing classes or methods.

1.7 Design Tools

The work products contained within this design are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the Chart II project, Release 1 configuration, System Design phase. A system version is included for each software package.

1.8 Work Products

This design contains the following work products:

- A UML Class diagram for each package showing the low level software objects which will allow the system to implement the interfaces identified in the high level design.
- UML Sequence diagrams for non-trivial operations of each interface identified in the high level design. Additionally, sequence diagrams are included for non-trivial methods in classes created to implement the interfaces. Operations that are considered trivial are operations that do nothing more than return a value or a list of values and where interaction between several classes is not involved.

2 Software Architecture

This section discusses various elements of the design that warrant more discussion than the UML diagrams afford. The High Level Design Document referenced above provides background information on CORBA and R1B1 Packaging and Deployment that may be necessary to fully benefit from the discussions below.

2.1 Service Application Framework

In a CORBA based system, service applications are used to serve CORBA objects through the ORB, making them available for use by other applications through a network. Once an object has been created and connected to the ORB, the object can act as an independent piece of software, given access to some basic services. The service applications that are built to serve CORBA objects usually share the same basic structure and functionality. The design team took advantage of this fact to provide a reusable framework for service applications.

The design of the application framework for CHART II CORBA Services is based upon two interfaces, the ServiceApplication and the ServiceApplicationModule. A class that implements the ServiceApplication interface is able to provide the basic services needed by CHART II CORBA objects. A ServiceApplicationModule is responsible for the initialization and shutdown of specific CORBA objects, using the services provided by the ServiceApplication.

A default implementation of the ServiceApplication interface is provided by the DefaultServiceApplication class. The DefaultServiceApplication is designed to meet the needs of all R1B1 service applications. Classes exist for each service application in the R1B1 CHART II system to provide a main entry point for the service application. As evidenced in the design, these service application classes do little more than construct a DefaultServiceApplication object and tell it to start, however their existence provides a place for service specific code should it be needed.

Several classes that implement the ServiceApplicationModule are included in this design, with each module responsible for serving one or more specific CHART II CORBA classes. Each of these modules has its own initialization and shutdown method tailored to the needs of the objects which it serves. Typical module initialization involves object creation from a state persisted in the database, connecting objects to the ORB, creation of an event channel, and publication of objects in the Trading Service. Typical module shutdown involves revoking offers from the Trading Service, destroying any previously created event channels, disconnecting objects from the ORB, and destroying the objects.

The DefaultServiceApplication is capable of hosting one or more ServiceApplicationModules. The modules served by a specific instance of the DefaultServiceApplication are specified by a configuration file used by the DefaultServiceApplication. This design allows for flexibility in the partitioning of objects among software processes. Modules can be brought together into a single process to achieve performance gains or moved to separate processes to provide greater fault isolation.

The design of the Service Application Framework is evidenced throughout this design. Packages exist for each module as well as the distinct service applications that will act as hosts for one or

more modules. Each service application package design includes a description of the modules that it will serve, and each module package design includes details on the CHART II CORBA objects which it will serve.

2.2 Event Channel Fault Tolerance

The standard CORBA event service contains a single event channel which is accessed through transient objects served by the event service called consumers and suppliers. Since the objects are transient, if the event service should crash, applications using the event service need to reinitialize their connection to the event service once it becomes available. The CHART II R1B1 design contains utility classes which allow applications to be tolerant of restarts of the event service. The PushEventSupplier, PushEventConsumer, and EventConsumerGroup classes, and the EventConsumer interface provide functionality for maintaining the connection to an event channel. The PushEventSupplier works as a wrapper to a CORBA PushSupplier which detects when an attempt to push fails and automatically attempts to reconnect on subsequent pushes.

The EventConsumer and EventConsumerGroup work together to allow multiple associations of event channels and consumers to be maintained, with a polling thread that periodically checks the connection of the consumer to the event channel and performs an automatic reconnect if necessary. The PushEventConsumer is an implementation of the EventConsumer that uses the push event model.

In addition to the need to provide fault tolerance for the CORBA Event Service, the event service's limitation to a single event channel causes events of all types to be passed on the same event channel. While this provides no hardship to suppliers of events, it requires consumers to filter the events to determine if they need to take action on an event or throw it away. This leads to inefficiency in both the processing required to filter the events as well as the network bandwidth used to pass unwanted events to consumers. This also makes it harder to provide a modular GUI design which allows seamless addition of new functionality.

To make up for this shortcoming, this design includes a package named the ExtendedEventService. This package specifies IDL for an EventChannelFactory interface that provides the capability for creating multiple event channels within a single EventService. The CHART II R1B1 design utilizes this added functionality to allow each module to be responsible for creating an event channel in their local event service and publishing the event channel object in the trader. This allows event channels throughout the system to be collected to provide a "big picture" of the real time status of the system and also provides fault isolation if an event service should fail.

2.3 Object Publication

As discussed in the High Level Design, the CORBA Trading Service is used by CHART II to allow CORBA objects to be discovered and used by other applications, including the CHART II GUI. All objects published in the Trading Service from CHART II applications are published with a service type equal to the interface name which the object implements. Full interface name hierarchies are used through the use of the supertypes registration feature (such as SharedResource / DMS) to allow generic as well as specific queries. All CHART II objects published in the trader have a standard mandatory property named "ID" of type octet sequence.

This ID is a globally unique identifier that remains with the object for the life of the object, even through multiple restarts of the service serving the object. Use of this ID allows objects to be located regardless of where they are being served in the system.

The following CHART II R1B1 objects are published in the Trading Service:

- Dictionary
- DMS
- DMSFactory
- DMSLibraryFactory
- DMSMessageLibrary
- DMSStoredMessage
- DMSStoredMsgItem
- Organization
- Plan
- PlanFactory
- UserManager

2.4 Database Access

A relational database is used to store system configuration data, persist object states (to allow restarts to assume their previous state), and to log user operations in the operations log. Java Database Connectivity (JDBC) is used within the application software to access the database. Access to the database is managed by the CHART II Database class. This class manages connections to the database. Each software package that requires access to the database includes a class that contains methods for all database accesses needed by the package. These classes are named with the package name and a suffix of DB. These database classes all use the Database object to obtain a connection to the database each time a series of queries or statements are to be executed. By managing a pool of actual database connections, the Database class makes sure that only one thread at a time has access to a given database connection, thus allowing transactional processing to be done safely.

2.5 Error Processing

Since CHART II is a distributed object system, it is expected that any call to a remote object could cause a CORBA exception to be thrown. All software calls to remote objects handle CORBA exceptions and the processing is not shown on sequence diagrams within this design except where it serves to illustrate a design point.

Furthermore, as with any system, most method calls, system calls, etc. can fail unexpectedly. All such errors are handled by the software and are not shown explicitly in the package design portion of this document. The default action when such an error is encountered is to reach a consistent state within the object where the error occurred and then to throw a

CHART2Exception (even for non-CORBA calls). The CHART2Exception contains debugging information as well as text suitable for display to a user or administrator. These exceptions are shown on sequence diagrams to call out error conditions that are not obvious.

Error conditions that involve throwing a specific exception as specified in the IDL are shown on sequence diagrams within this design.

2.6 Service Application Maintenance

Although not a requirement of R1B1, all service applications implement the IDL Service interface to allow for clean service shutdown. In addition to allowing shutdown, the Service interface includes features that will be useful for a future system monitor process. These features include the ability for a service to tell its name when asked, tell the network connection site where it is running, and respond to a ping operation. Since the Service is a CORBA object attached to an ORB, these operations on a service can be accessed from anywhere on the CHART II network.

2.7 Packaging

This software design is broken into many packages of related classes. The table below shows each of the packages along with a description of each.

CORBAUtilities	This package contains classes included in the third party ORB product used for implementation. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
JavaClasses	This package contains classes included in the Java programming language. Only classes that are directly referenced from diagrams for CHART II software are included in this package's diagrams.
DictionaryModule	This package contains classes needed to implement the Dictionary interface as specified in the high level design.
DMSControlModule	This package contains classes necessary for implementing the DMSFactory, DMS, and DMSStoredMsgItem interfaces as specified in the high level design.
DMSLibraryModule	This package contains classes necessary for implementing the DMSLibraryFactory, DMSMessageLibrary, and StoredDMSMessage interfaces specified in the high level

	design.
DMSService	This package contains classes needed to provide an executable host to the R1B1DMSControlModule, R1B1DMSLibraryModule, and R1B1DictionaryModule.
ExtendedEventService	This package contains classes used to extend the event service provided by the ORB vendor to allow multiple event channels to be created. This is done through the definition and implementation of the EventChannelFactory interface.
PlanModule	This package contains classes needed to implement the PlanFactory, Plan, and Plan Item interfaces specified in the high level design.
PlanService	This package contains classes necessary to provide an executable host to the R1B1PlanModule.
SystemInterfaces	This package contains the interfaces specified in the high level design. Each class in this package is also specified in IDL and can be accessed in the system using CORBA.
UserManagementModule	This package contains classes necessary to implement the UserManager interface specified in the high level design.
UserManagement ResourcesModule	This package contains classes necessary to implement the OperationsCenter and Organization interfaces specified in the high level design.
UserManagementService	This package contains classes necessary to provide an executable host to the R1B1UserManagementModule and the R1B1UserManagementResourcesModule.
Utility	This package contains utility classes shared by other packages, including classes used to access the database and the OperationsLog class.

The remainder of this document contains detailed designs of each of the above packages.

3 Package Designs

The following sections provide detailed designs of each of the software packages included in CHART II R1B1. Each section contains a class diagram and sequence diagrams for non-trivial operations identified in the High Level Design IDL.

3.1 DMSService

3.1.1 DMSServiceClasses (Class Diagram)

The DMSService is an application that publishes objects relating to Dynamic Message Signs (DMS). The service itself contains minimal functionality and serves as a host to modules that it installs based on properties it reads at runtime. It is these modules that actually serve and publish the objects that provide the functionality of the DMS Service Application.

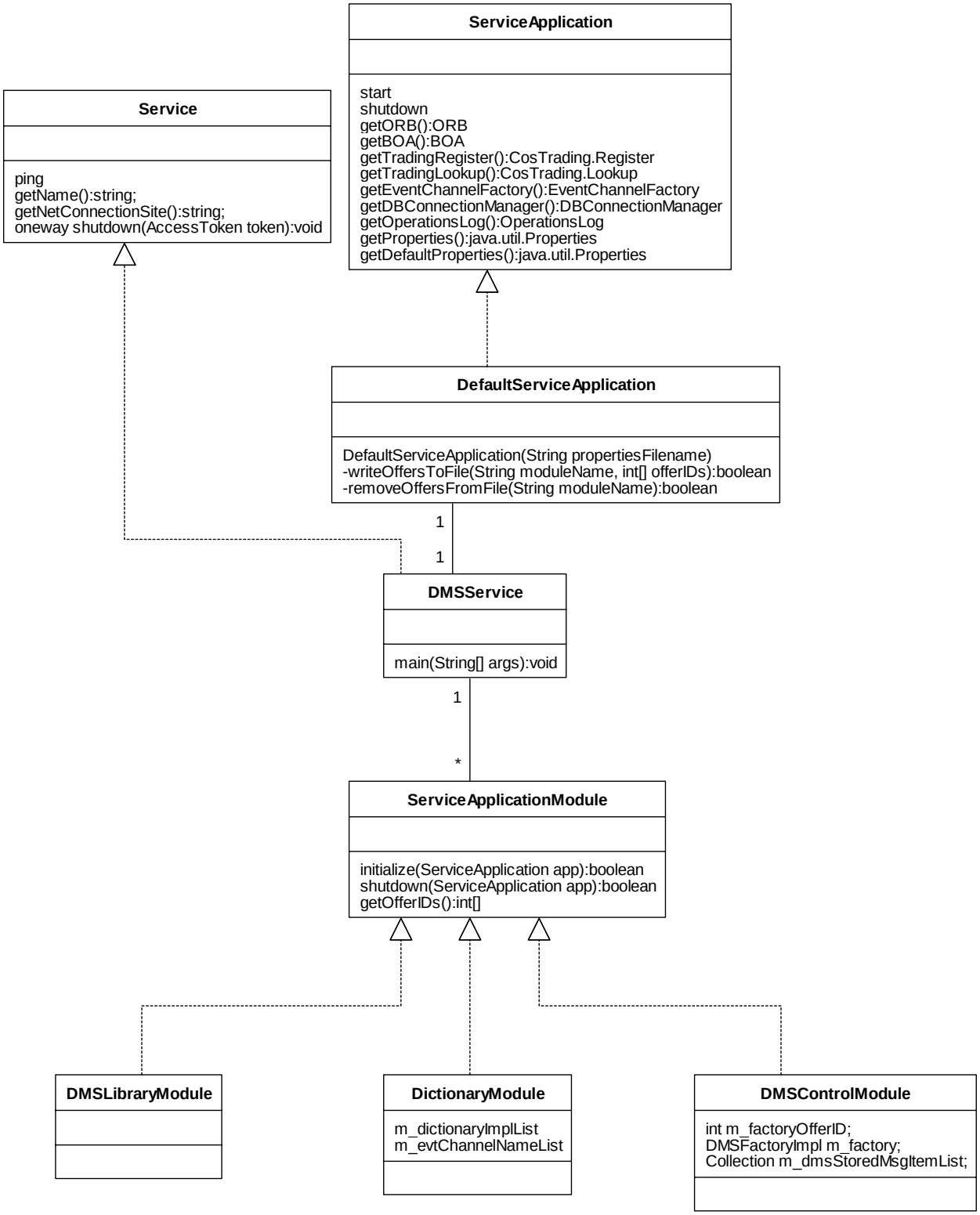

Figure 3-1. DMSServiceClasses (Class Diagram)

3.1.1.1 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.1.1.2 DMSControlModule (Class)

This class implements the ServiceApplicationModule interface. It creates and serves a single DMSFactoryImpl object, which in turn serves zero or more DMSImpl objects. This module also serves DMSStoredMsgItemImpl objects that were created for DMSImpls being served from this module.

3.1.1.3 DictionaryModule (Class)

This class implements the Service Application module interface. It publishes the dictionary implementation.

3.1.1.4 DMSLibraryModule (Class)

This module manages the Message Libraries and Stored Messages for the DMS service. It provides the functionality to add, delete and modify the libraries and messages stored in them.

3.1.1.5 DMSService (Class)

This class provides the main method for the DMS Service Application. It acts as the host for the DMS Control, DMS Library, and Dictionary server modules. It makes use of the DefaultServiceApplication to provide access to standard objects to the server modules.

3.1.1.6 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.1.1.7 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

interface

3.1.1.8 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.1.2 Sequence Diagrams

3.1.2.1 DMSService:Shutdown (Sequence Diagram)

This sequence diagram shows the shutdown of the DMS Service module. It disconnects itself from the ORB, then calls the DefaultServiceApplication object to shutdown. Refer to the Utility package's DefaultServiceApplication:Shutdown sequence diagram for details on the shutdown of the DefaultServiceApplication.

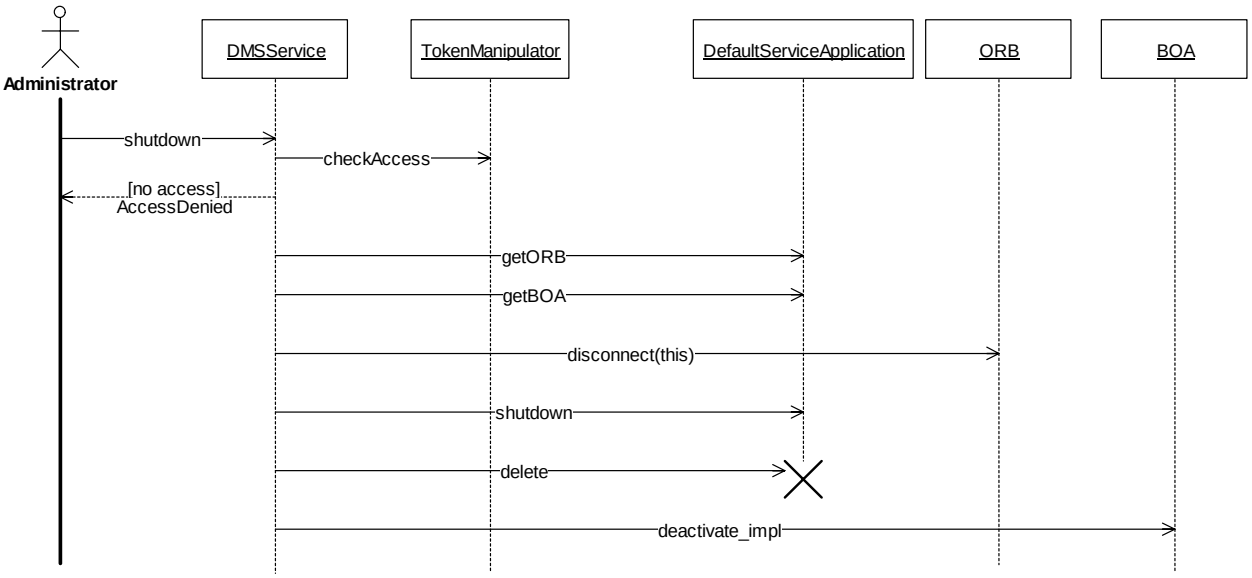

Figure 3-2. DMSService:Shutdown (Sequence Diagram)

3.1.2.2 DMSService:Startup (Sequence Diagram)

This sequence shows the startup of the DMS Service. This service acts as the host for the DMS Control Module, the DMS Library Module, and the Dictionary module for R1B1, however the modules served is configurable by an initialization file. During startup, this service simply creates a DefaultServiceApplication object passing it the name of the properties file for the service. The DMSService then starts the DefaultServiceApplication, which initializes common services, such as the database, ORB and CORBA services. The DefaultServiceApplication also creates and initializes each of the modules configured in the properties file. After the DefaultServiceApplication has started, the DMSService connects itself to the ORB, since it implements the Service interface, and then notifies the BOA to start accepting CORBA requests.

Refer to the Utility package's DefaultServiceApplication:Start sequence diagram for details on the startup of the DefaultServiceApplication. Refer to the DMSControlModule, DMSLibraryModule, and DictionaryModule detailed designs for information on initialization of these modules.

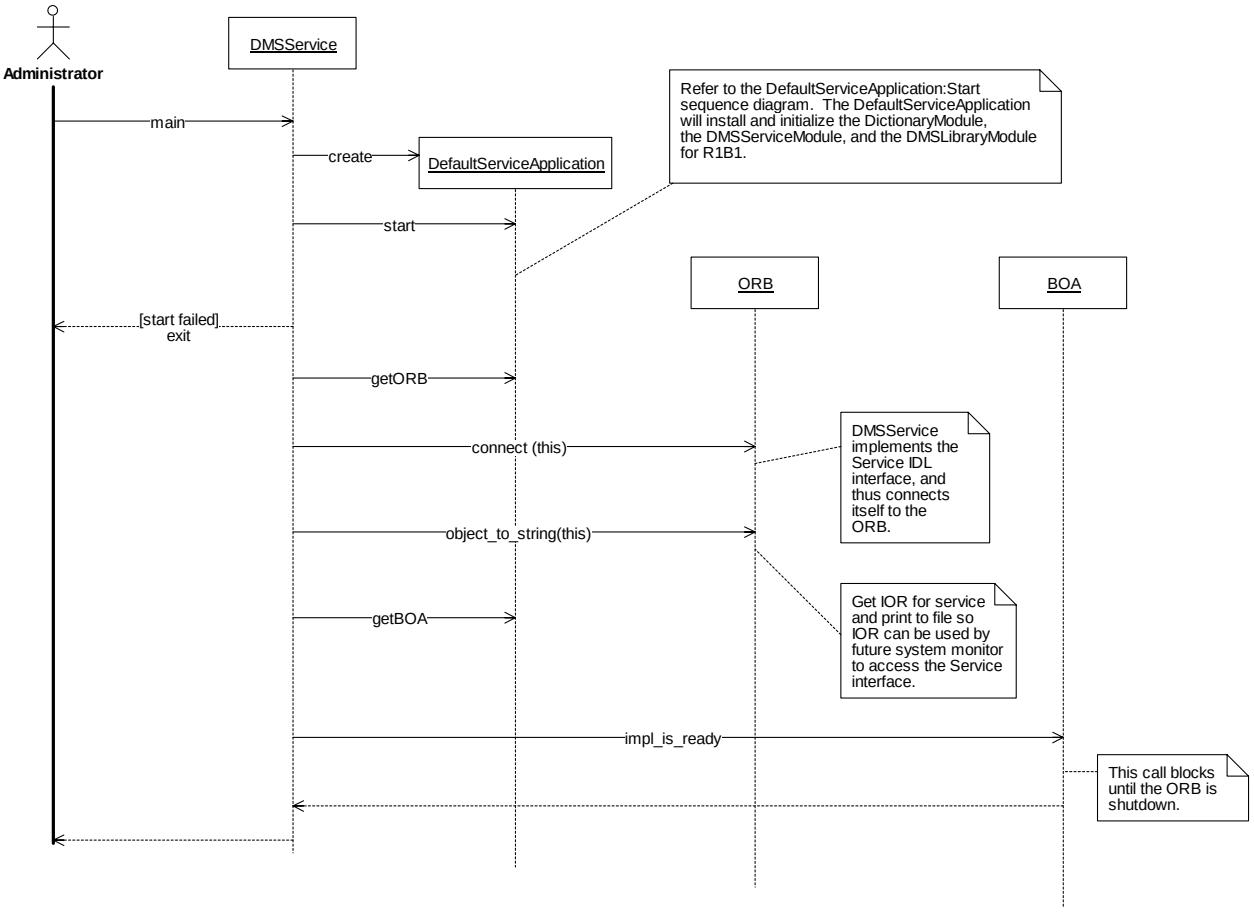

Figure 3-3. DMSService:Startup (Sequence Diagram)

3.2 DMSControlModule

3.2.1 DMSControlClassDiagram (Class Diagram)

The DMSControlModule is an installable module that serves the DMSFactory, DMS, and DMSStoredMsgItem objects to the rest of the Chart2 system. This diagram shows how the implementation of these CORBA interfaces rely on other supporting classes to perform their functions.

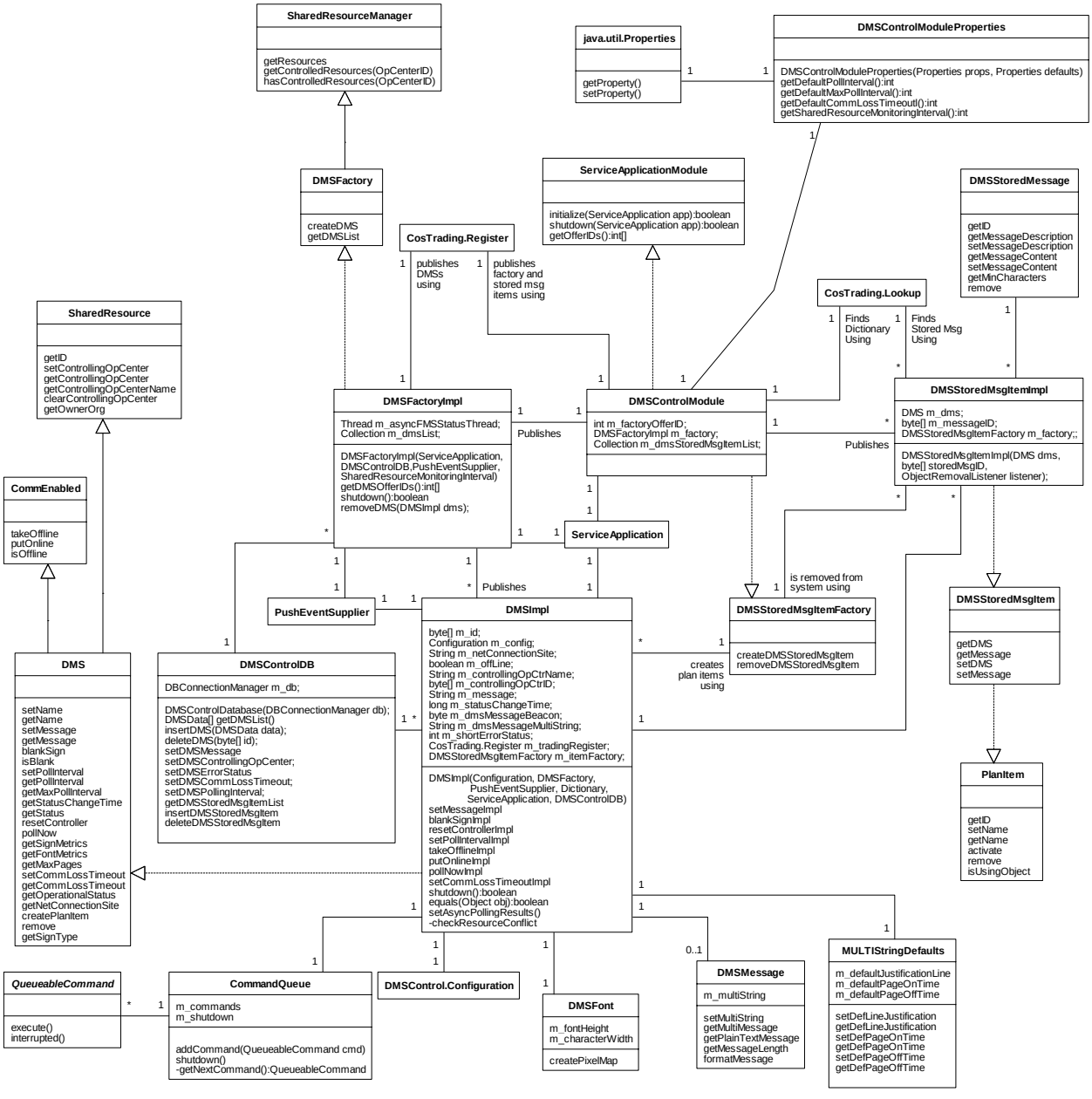

Figure 3-4. DMSControlClassDiagram (Class Diagram)

3.2.1.1 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can have their communications turned on or off. This typically only applies to field devices.

1

interface

3.2.1.2 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

1

interface

3.2.1.3 DMS (Class)

This class represents a Dynamic Message Sign (DMS). It has attributes and methods for controlling and maintaining the status of the DMS within the system.

interface

3.2.1.4 DMSFactoryImpl (Class)

The DMSFactoryImpl class provides an implementation of the DMSFactory interface as specified in the IDL. The DMSFactory maintains a list of DMSImpl objects and is responsible for publishing DMS objects in the Trader. It maintains a mapping of offer ids received from the trader for each DMS object published so that it may withdraw the offers during shutdown or when a DMS is removed from the system.

3.2.1.5 DMSControl.Configuration (Class)

This typedef defines data that is used to identify the configuration of a DMS in the system.

1

typedef

3.2.1.6 DMSControlModule (Class)

This class implements the ServiceApplicationModule interface. It creates and serves a single DMSFactoryImpl object, which in turn serves zero or more DMSImpl objects. This module also serves DMSStoredMsgItemImpl objects that were created for DMSImpls being served from this module.

3.2.1.7 DMSControlModuleProperties (Class)

This class is used to provide access to properties used by the DMS Control Module. This class wraps properties that are passed to it upon construction. It adds its own defaults and provides methods to extract properties specific to the DMS Control Module.

3.2.1.8 DMSControlDB (Class)

The DMSControlDB class is a collection of methods that perform database operations on tables pertinent to DMS Control. The class is constructed with a Database object, which manages database connections. Every operation in this class obtains a connection to the database from the Database object prior to performing the DB operation which is requested.

3.2.1.9 DMSFactory (Class)

The DMSFactory provides a means to create new DMS objects to be added to the system.

1

interface

3.2.1.10 DMSFont (Class)

This class contains the functionality for translating text messages into pixels for display on a DMS.

utility

3.2.1.11 DMSImpl (Class)

The DMSImpl class implements the DMS, SharedResource, and CommEnabled interfaces specified by IDL. The DMSImpl contains a command queue that is used to execute long running operations in a thread separate from the CORBA request threads, thus allowing quick initial responses. The DMSImpl contains *Impl methods that map to each method specified in the IDL that requires command queuing. The queueable command objects simply call the appropriate DMSImpl method as the command is executed by the command queue in the queue's thread of execution.

3.2.1.12 DMSMessage (Class)

This utility class represents a text message which is capable of being stored on a DMS. It contains methods for input and output of the message in different formats.

utility

3.2.1.13 QueueableCommand (Class)

A QueuableCommand is an abstract class used to represent a command that can be placed on a queue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed.

1

3.2.1.14 DMSStoredMessage (Class)

This class represents a stored DMS message which is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

interface

3.2.1.15 MULTIStringDefaults (Class)

This class contains the model-specific default values for creating MULTI strings for a DMS. MULTI is a standard mark-up language specified by NTCIP for specifying how a text message is to be displayed by a DMS.

3.2.1.16 DMSStoredMsgItem (Class)

This class represents a plan item that is used to associate a stored DMS message with a specific DMS. When the item is activated, it sets the message of the DMS to the stored message to which it is linked.

interface

3.2.1.17 DMSStoredMsgItemImpl (Class)

This class implements the DMSStoredMsgItem interface as defined in IDL. It acts as an association between a DMS and a StoredDMSMessage.

3.2.1.18 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

interface

3.2.1.19 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

1

interface

3.2.1.20 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center.

1

interface

3.2.1.21 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

1

3.2.1.22 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.2.1.23 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects which have previously been published.

1

interface

3.2.1.24 DMSStoredMsgItemFactory (Class)

This interface is implemented by objects that can act as a factory for DMSStoredMsgItem objects. Implementing classes must know how to create a DMSStoredMsgItem and add it to the system and how to remove a DMSStoredMsgItem from the system.

interface

3.2.1.25 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.2.1.26 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

3.2.1.27 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

1

interface

3.2.2 QueueableCommandClassDiagram (Class Diagram)

This class diagram shows the classes that are derived from QueueableCommand. A class exists for each type of command that can be executed asynchronously on a DMS object.

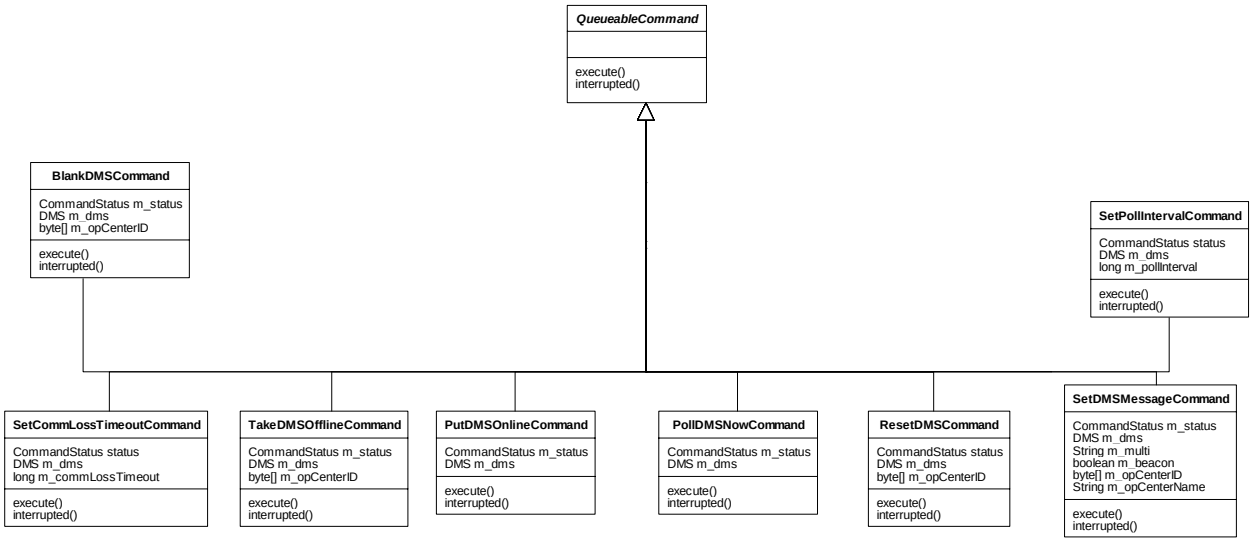

Figure 3-5. QueueableCommandClassDiagram (Class Diagram)

3.2.2.1 PollDMSNowCommand (Class)

This class is used as a holder for command data needed to execute the pollNow method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object that it has stored.

3.2.2.2 PutDMSOnlineCommand (Class)

This class is used as a holder for command data needed to execute the putDMSOnline method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object that it has stored.

3.2.2.3 BlankDMSCommand (Class)

This class is used as a holder for command data needed to execute the blankSign method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object that it has stored.

3.2.2.4 QueueableCommand (Class)

A QueuableCommand is an abstract class used to represent a command that can be placed on a queue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed.

1

3.2.2.5 SetCommLossTimeoutCommand (Class)

This class is used as a holder for command data needed to execute the setCommLossTimeout method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object using the parameters that it has stored.

3.2.2.6 SetDMSMessageCommand (Class)

This class is used as a holder for command data needed to execute the setMessage method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object passing the parameters that it has stored.

3.2.2.7 SetPollIntervalCommand (Class)

This class is used as a holder for command data needed to execute the setPollInterval method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object using the parameters that it has stored.

3.2.2.8 ResetDMSCommand (Class)

This class is used as a holder for command data needed to execute the resetController method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object that it has stored.

3.2.2.9 TakeDMSOfflineCommand (Class)

This class is used as a holder for command data needed to execute the takeDMSOffline method of the DMSImpl. This object is placed on a queue and executed in a separate thread of execution. When executed, it calls the appropriate method on the DMSImpl object that it has stored.

3.2.3 Sequence Diagrams

3.2.3.1 DMSControlModule:ActivateDMSStoredMsgItem (Sequence Diagram)

A DMSStoredMsgItem is served from within the DMS Control Module, however it is referenced as part of a Plan under the guise of a generic plan item. When a DMSStoredMsgItem's activate method is called, the item gets the contents of its stored message and uses the contents to set the message on its associated DMS. Since the DMSStoredMsgItem stores references to CORBA objects, there is the possibility that either the StoredDMSMessage or DMS are not available (they may exist on different servers etc.) When this occurs, the DMSStoredMessage Item cannot activate and updates the command status to indicate its failed completion status.

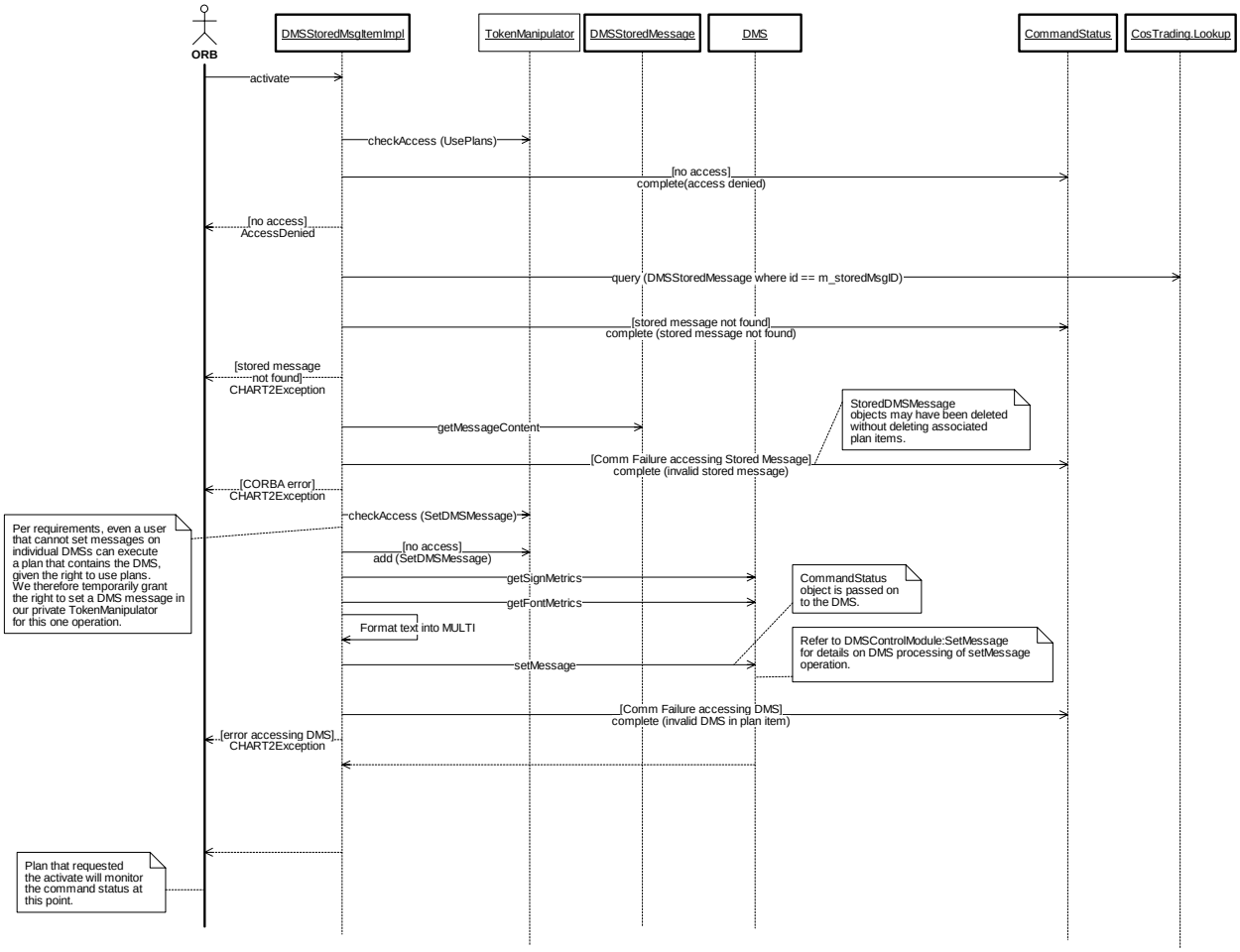

Figure 3-6. DMSControlModule:ActivateDMSStoredMsgItem (Sequence Diagram)

3.2.3.2 DMSControlModule:BlankSign (Sequence Diagram)

This sequence diagram shows the required operations to blank a DMS. Since field communications are involved, the actual blanking of the DMS is performed in a separate thread using a command queue. Prior to queing the command, preliminary checks are done to ensure the request can be completed given operational rules. A Command Status object is used by the caller to track the progress of the asynchronous command. Events are pushed on an event channel after the sign is blanked and when the controlling operations center has been cleared.

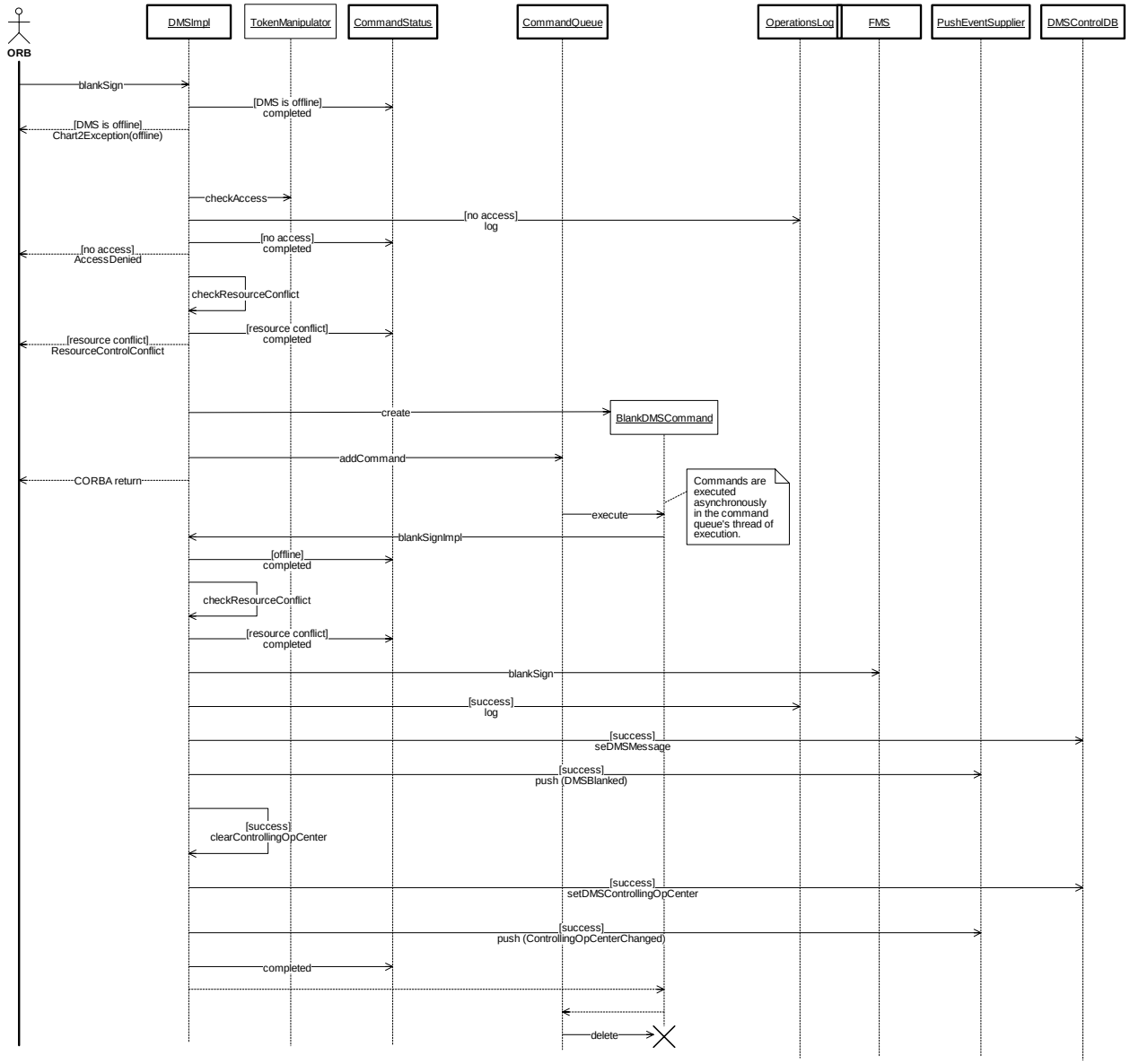

Figure 3-7. DMSControlModule:BlankSign (Sequence Diagram)

3.2.3.3 DMSControlModule:CheckResourceConflict (Sequence Diagram)

This sequence diagram shows how a DMS determines if a user from a given operations center should be allowed to perform an operation that will cause the controlling operations center for the DMS to be set. This is used by all operations that affect the display of the sign.

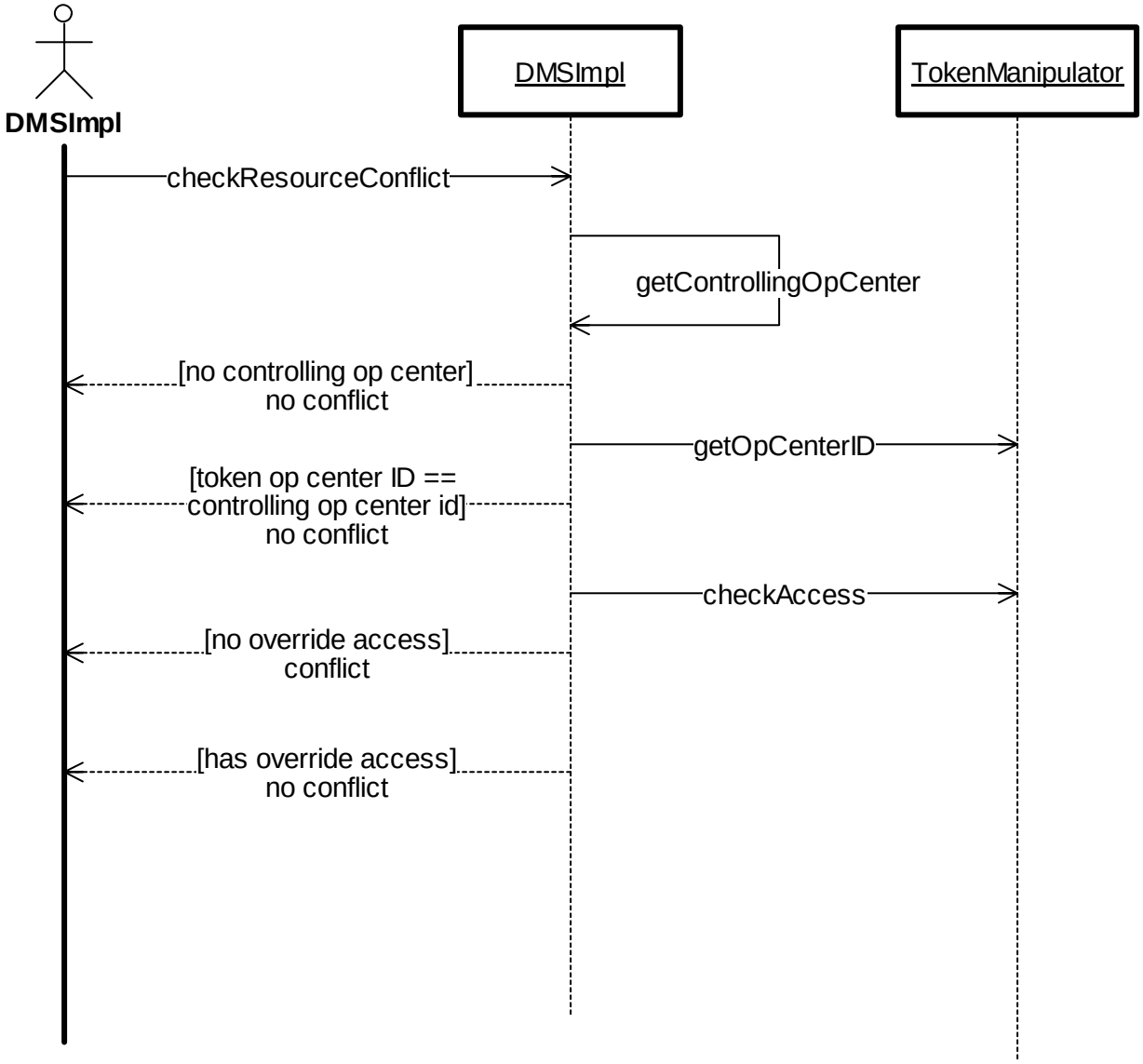

Figure 3-8. DMSControlModule:CheckResourceConflict (Sequence Diagram)

3.2.3.4 DMSControlModule:DMSControlModuleCreatePlanItem (Sequence Diagram)

The DMSControlModule class implements the DMSStoredMsgItemFactory interface and as such is responsible for creating DMSStoredMsgItem objects when requested. This class is responsible for creating the DMSStoredMsgItemImpl object and connecting it to the ORB to make it available through CORBA. The object's data is persisted to the database and the object is published in the Trading Service. The publication in the trader allows the plan which contains the plan item to locate the object no matter where the object is being served.

(Since a DMSStoredMsgItem is specific to DMS control, the DMSControlModule serves DMSStoredMsgItem objects. Likewise, plans are served from the PlanModule, thus the Plan object that contains the DMSStoredMsgItem is served from a different service application. The Plan can find the object references for its items in the trader so that it can activate the plan items.)

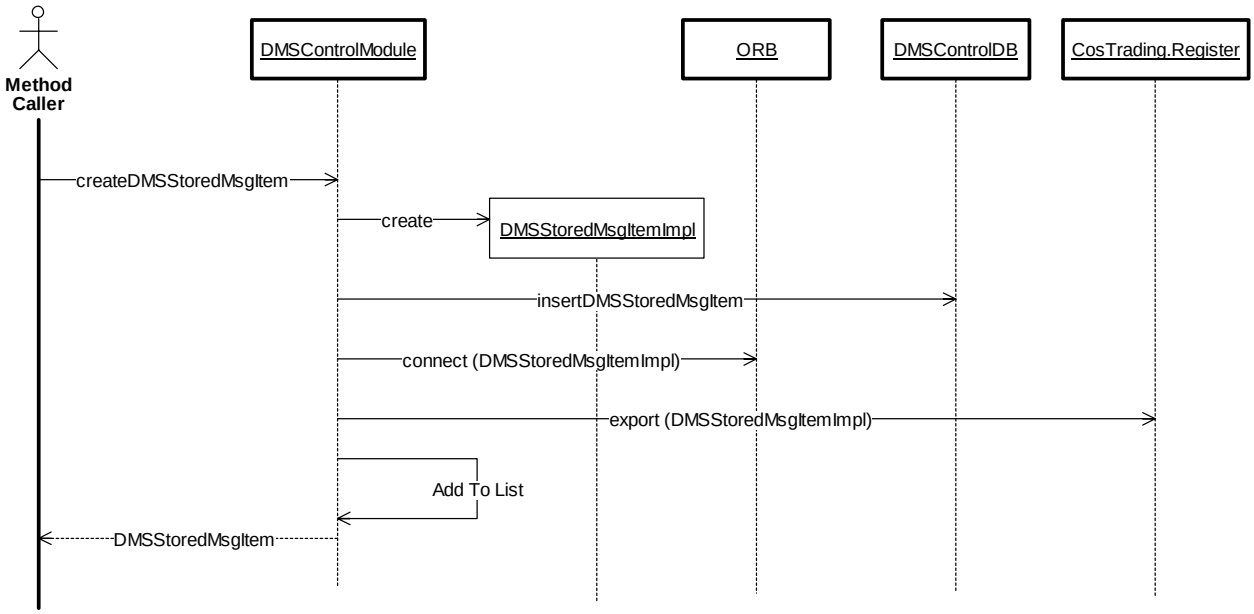

Figure 3-9. DMSControlModule:DMSControlModuleCreatePlanItem (Sequence Diagram)

3.2.3.5 DMSControlModule:DMSControlModuleRemovePlanItem (Sequence Diagram)

The DMSControlModule class implements the DMSStoredMsgItemFactory interface and as such is responsible for deleting DMSStoredMsgItem objects from the system when requested. The offer of the DMSStoredMsgItem is withdrawn from the trader, the DMSStoredMsgItem data is removed from the database, and the object is disconnected from the ORB.

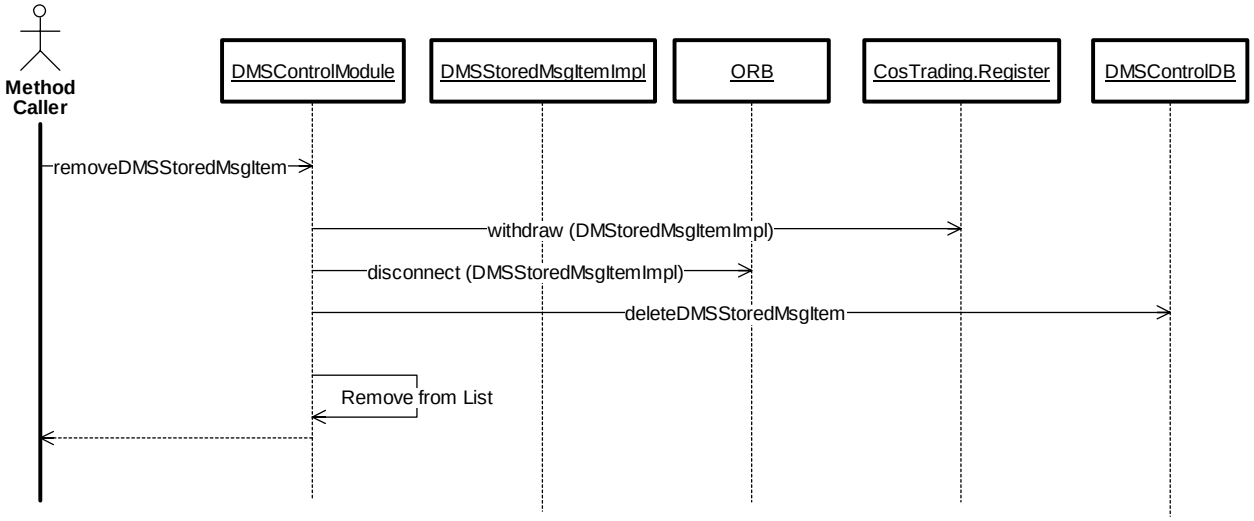

Figure 3-10. DMSControlModule:DMSControlModuleRemovePlanItem (Sequence Diagram)

3.2.3.6 DMSControlModule:GetControlledResources (Sequence Diagram)

This sequence diagram shows how a DMS Factory reports the resources it contains that are under the control of a specific operations center. The factory simply asks each DMS object for its controlling operations center and adds the DMS to a list if the operations center matches the operations center in question. The list of DMSs under the control of the given operations center is then returned.

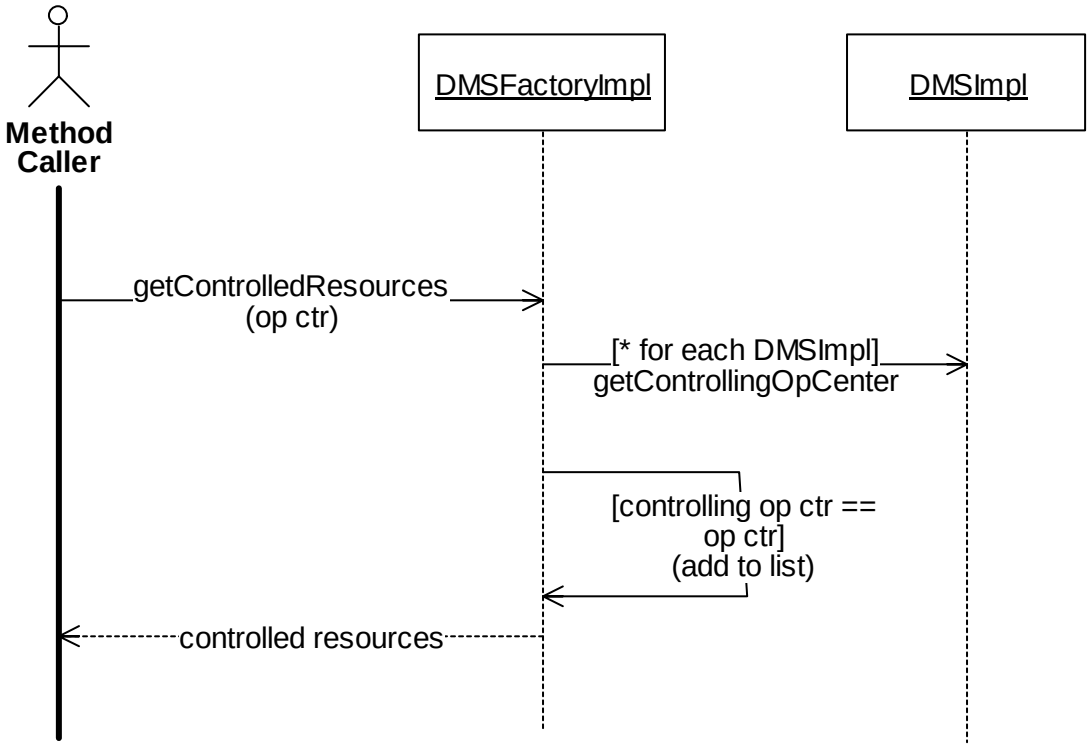

Figure 3-11. DMSControlModule:GetControlledResources (Sequence Diagram)

3.2.3.7 DMSControlModule:HasControlledResources (Sequence Diagram)

This sequence shows how a DMS Factory reports whether or not any of the DMSs in the factory are currently being controlled by a given operations center. The method returns only true or false, so as soon as one DMS under the control of the operations center is found the method can return true without looking further.

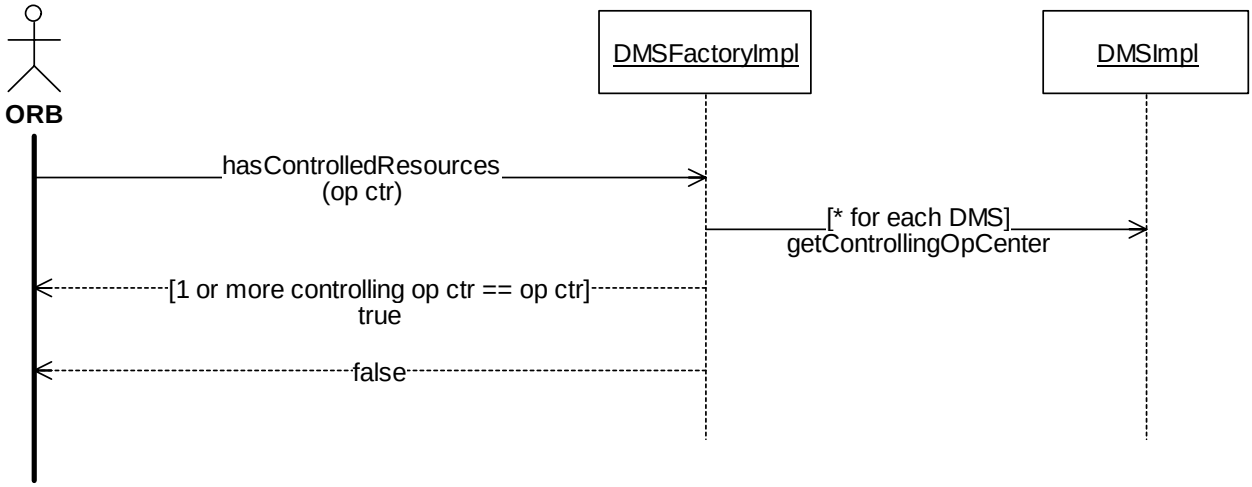

Figure 3-12. DMSControlModule:HasControlledResources (Sequence Diagram)

3.2.3.8 DMSControlModule:Initialize (Sequence Diagram)

This sequence diagram shows the startup for the DMS Control Module. This module is created by a service that will host this module's objects. A ServiceApplication is passed to this module's initialize method and provides access to basic objects needed by this module. This module creates a DMS Factory which in turn creates DMS objects. In addition, the module creates and serves objects for any plan items that were previously created for each DMS. The DMSFactory and DMS objects are published via the CORBA Trading service to make them available for general status updates and candidates for control (given the proper access rights).

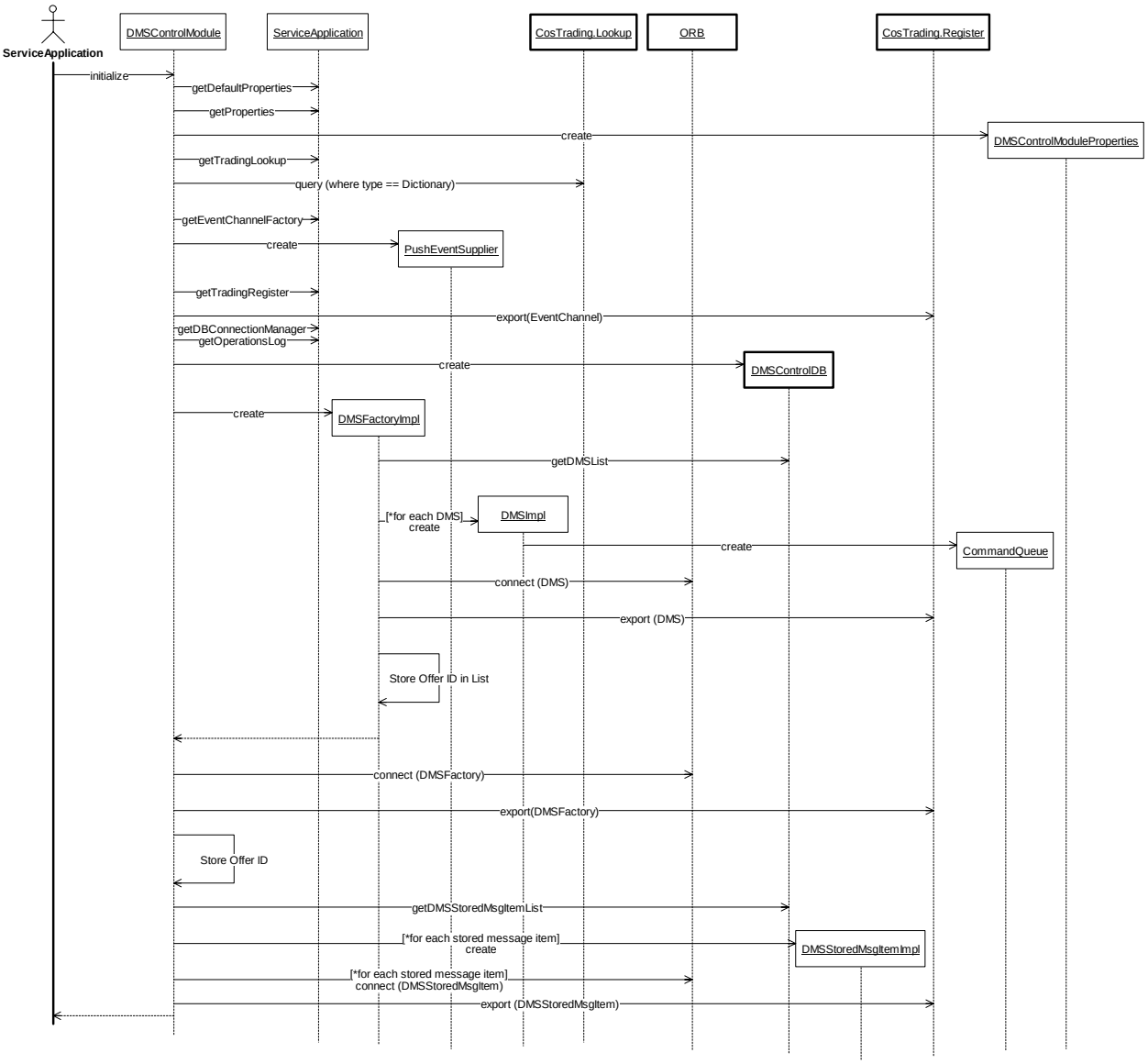

Figure 3-13. DMSControlModule:Initialize (Sequence Diagram)

3.2.3.9 DMSControlModule:MonitorControlledResources (Sequence Diagram)

This sequence diagram shows how the DMSFactory monitors its controlled resources to detect when a DMS is left displaying a message with no one logged into the operations center that has control of the DMS. During the creation of the DMSFactoryImpl, a thread is created that is used to periodically get a summarized list of all operations centers that are controlling one or more DMS's contained in the factory. Each operations center is then checked to make sure there are 1 or more users logged in. When the condition exists where the operations center does not have at least one user logged in but is in control of 1 or more DMSs, an alarm is raised through the event channel.

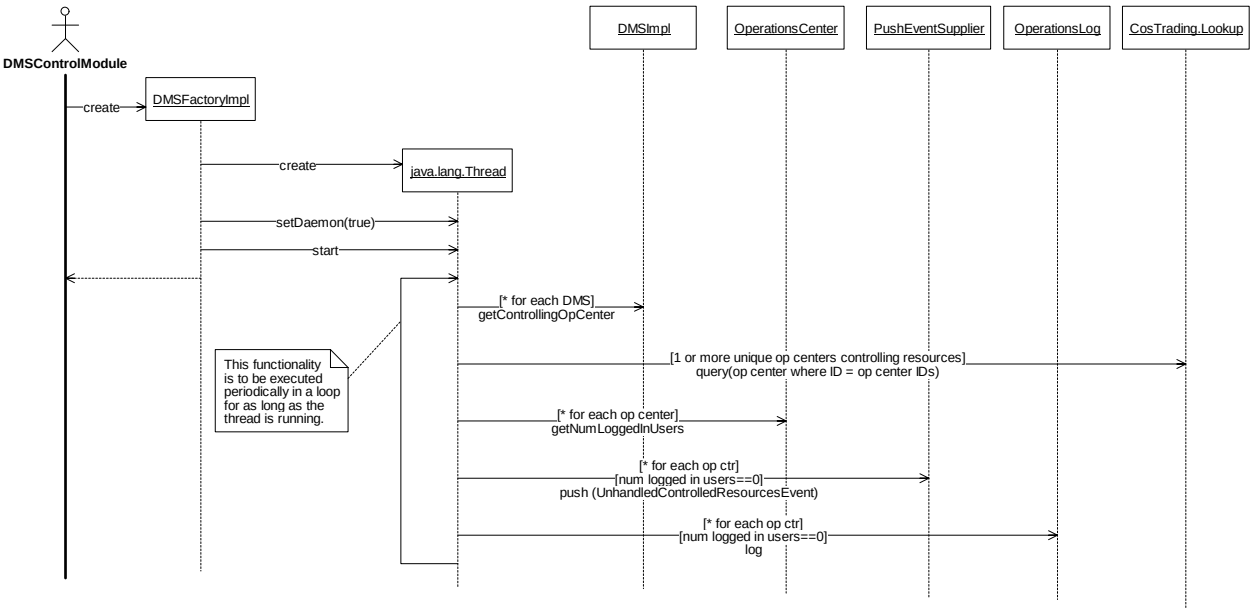

Figure 3-14. DMSControlModule:MonitorControlledResources (Sequence Diagram)

3.2.3.10 DMSControlModule:ProcessFMSPollingResults (Sequence Diagram)

The FMS Subsystem polls each DMS in the system and supplies discrepancies or problems found during the polling through a blocking call. For this reason, the DMSFactoryImpl contains a thread that is used to execute this blocking call. When the blocking call returns, the DMSFactoryImpl dispatches the changed status or error condition to the proper DMSImpl object so that it may update its internal state and push an event through the event service.

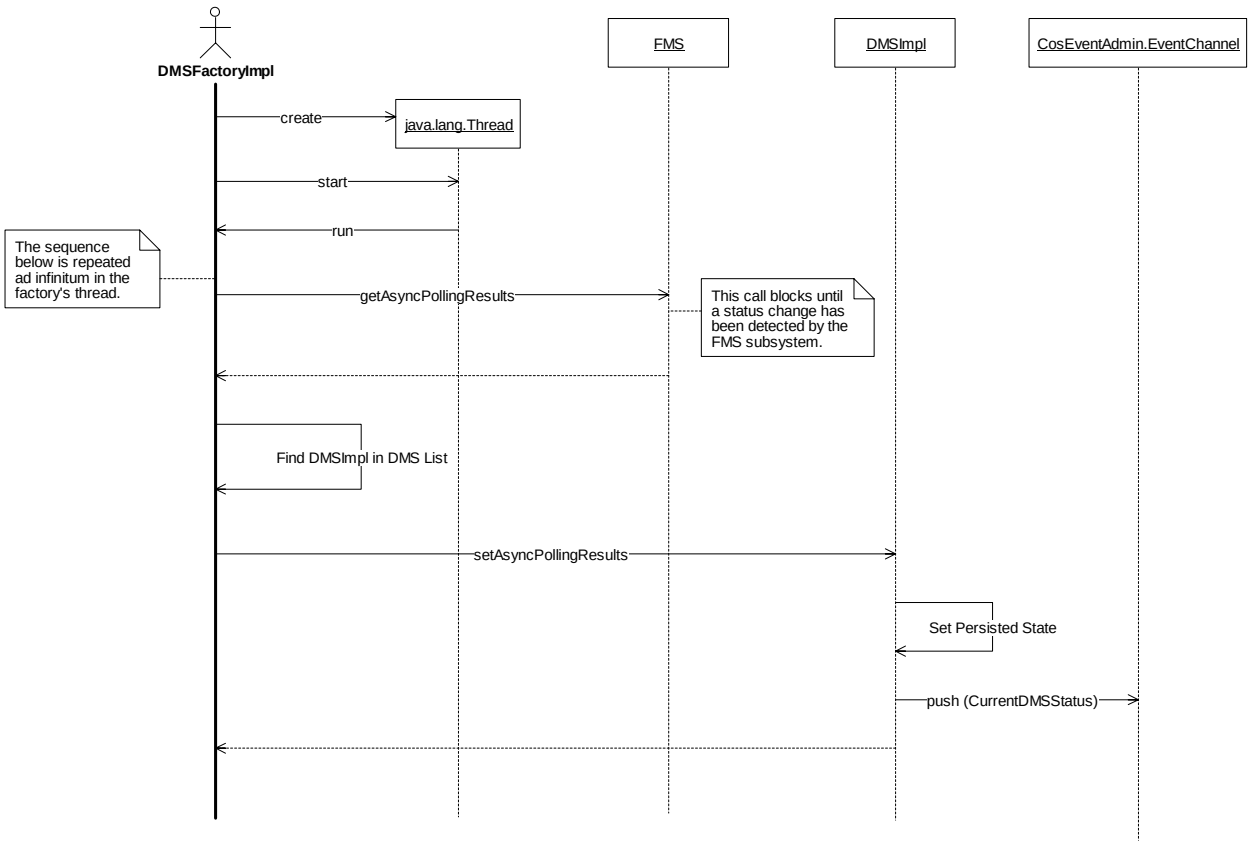

Figure 3-15. DMSControlModule:ProcessFMSPollingResults (Sequence Diagram)

3.2.3.11 DMSControlModule:RemoveDMS (Sequence Diagram)

This sequence diagram shows the processing done by the DMSFactoryImpl when its objectRemoved method is called. This metod is defined in the ObjectRemoval interface and is called by a DMSImpl when it is removed from the system, giving the DMSFactoryImpl a chance to clean up any references it may have to the DMSImpl. The Factory must remove the reference to the DMSImpl from its internal list of DMSs, remove the DMSImpl from the database and the FMS subsystem, and withdraw the DMS's offer from the trading service.

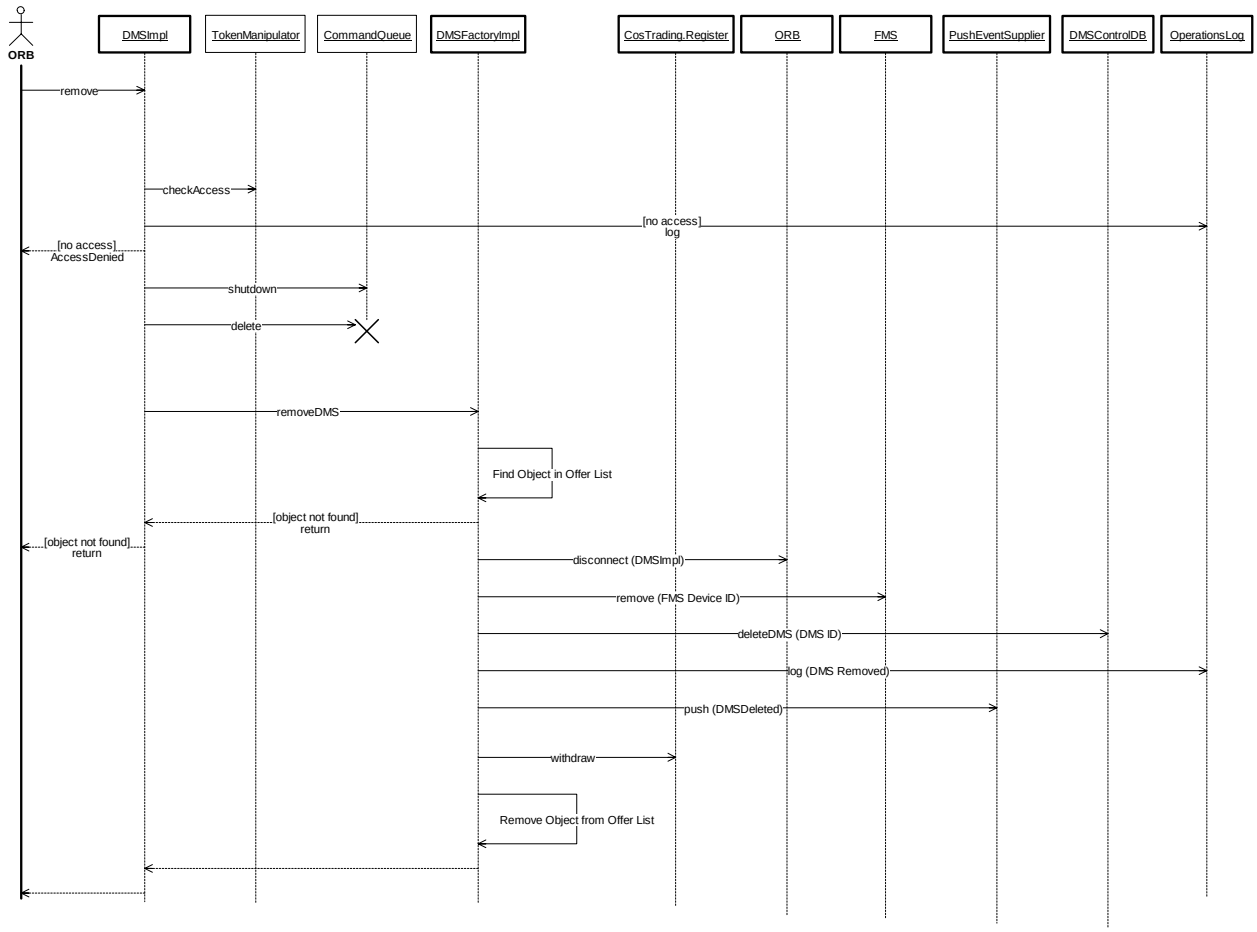

Figure 3-16. DMSControlModule:RemoveDMS (Sequence Diagram)

3.2.3.12 DMSControlModule:RemoveDMSStoredMsgItem (Sequence Diagram)

When a DMSStoredMsgItem is removed from the system, the DMSStoredMsgItem delegates the call to a DMSStoredMsgItemFactory that was passed to the DMSStoredMsgItem during construction. The factory takes care of cleaning up trader offers, database records, etc. Refer to the DMSControlRemovePlanItem sequence diagram for details.

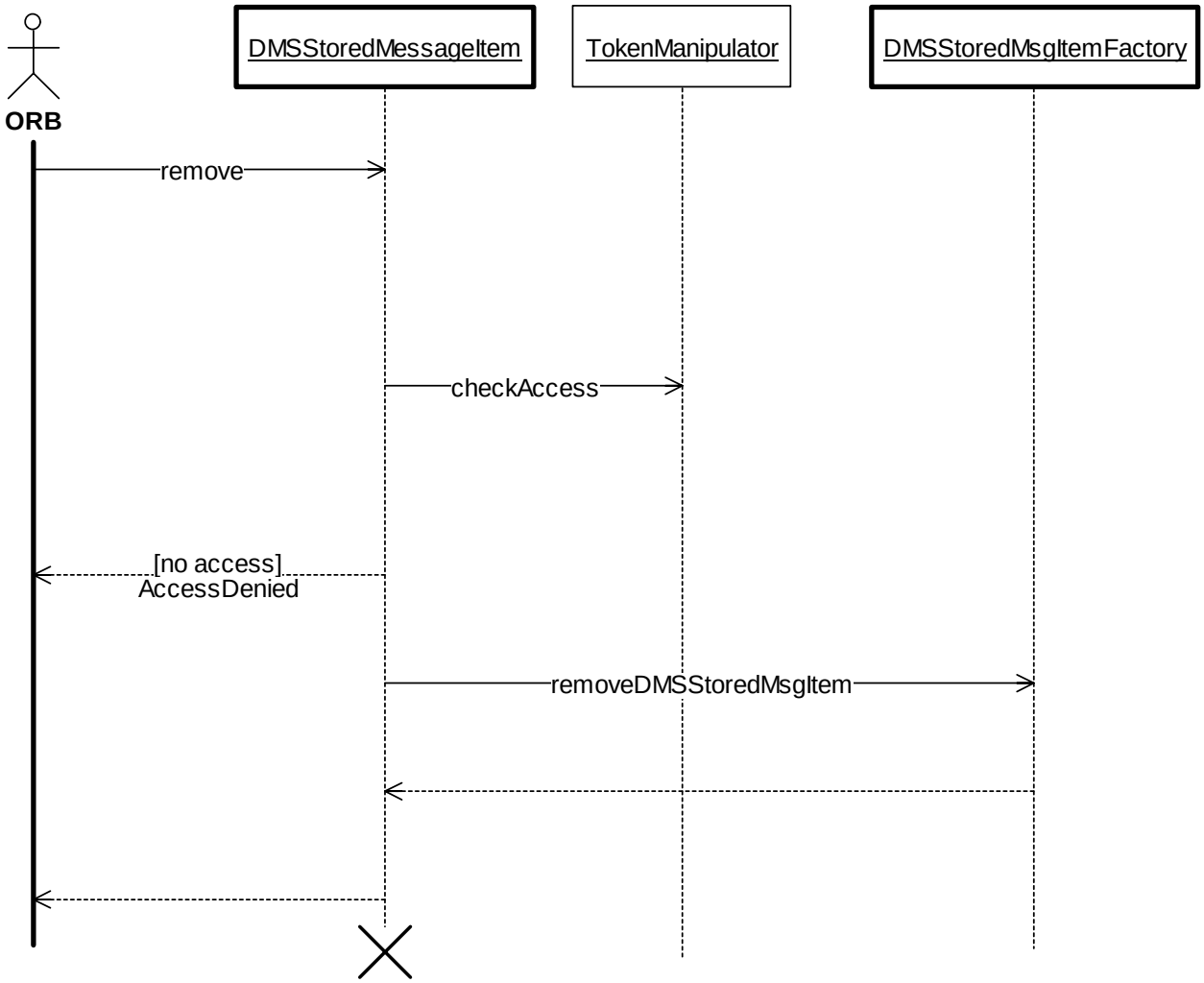

Figure 3-17. DMSControlModule:RemoveDMSStoredMsgItem (Sequence Diagram)

3.2.3.13 DMSControlModule:CreateDMS (Sequence Diagram)

When a DMS is added to the DMS factory, the default values are read from the DMSControlModuleProperties and the DMS data is added to the database. A DMSImpl object and its corresponding command queue is created and the object is connected to the ORB. The DMSAdded event is then pushed into the event channel. Access is denied if the caller does not possess requisite privilege(s).

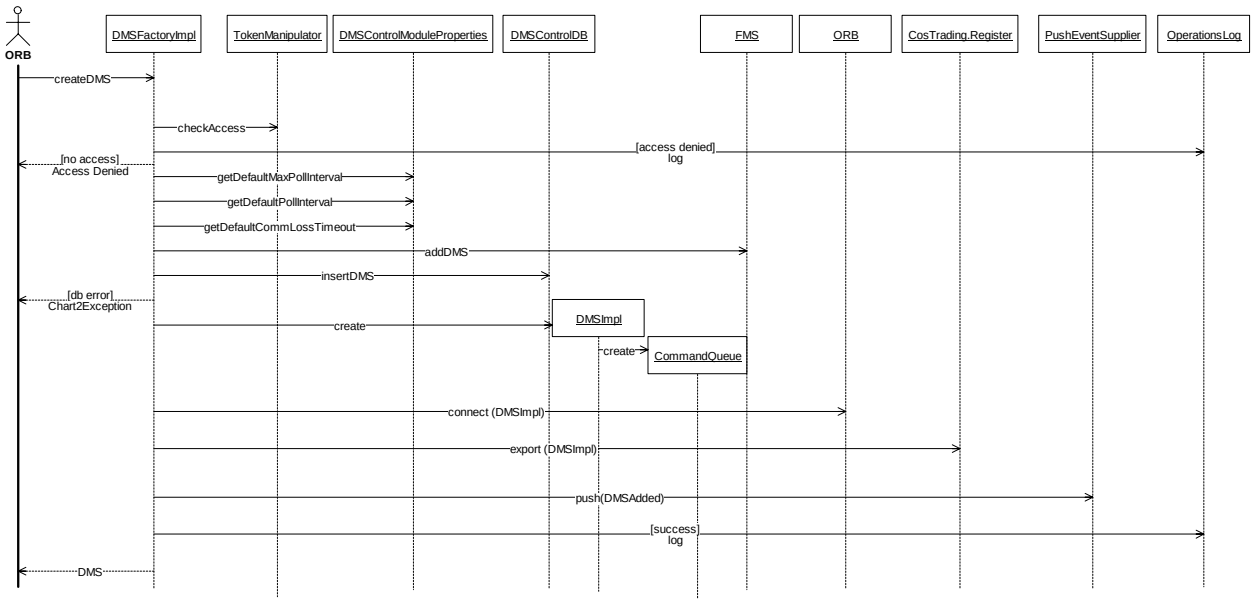

Figure 3-18. DMSControlModule:CreateDMS (Sequence Diagram)

3.2.3.14 DMSControlModule:CreatePlanItem (Sequence Diagram)

The DMS Module serves all DMSStoredMsgItem objects. These objects associate a stored DMS message with a DMS. When a DMS is asked to create a plan item, it passes the call onto a DMSStoredMsgItem factory which it was passed during construction. The DMS object serves only as a convenient creation point for DMS related plan items. It is actually the implementer of the DMSStoredMsgItemFactory interface that manages the collection of DMSStoredMsgItem objects. Refer to the DMSControlCreatePlanItem sequence diagram for details.

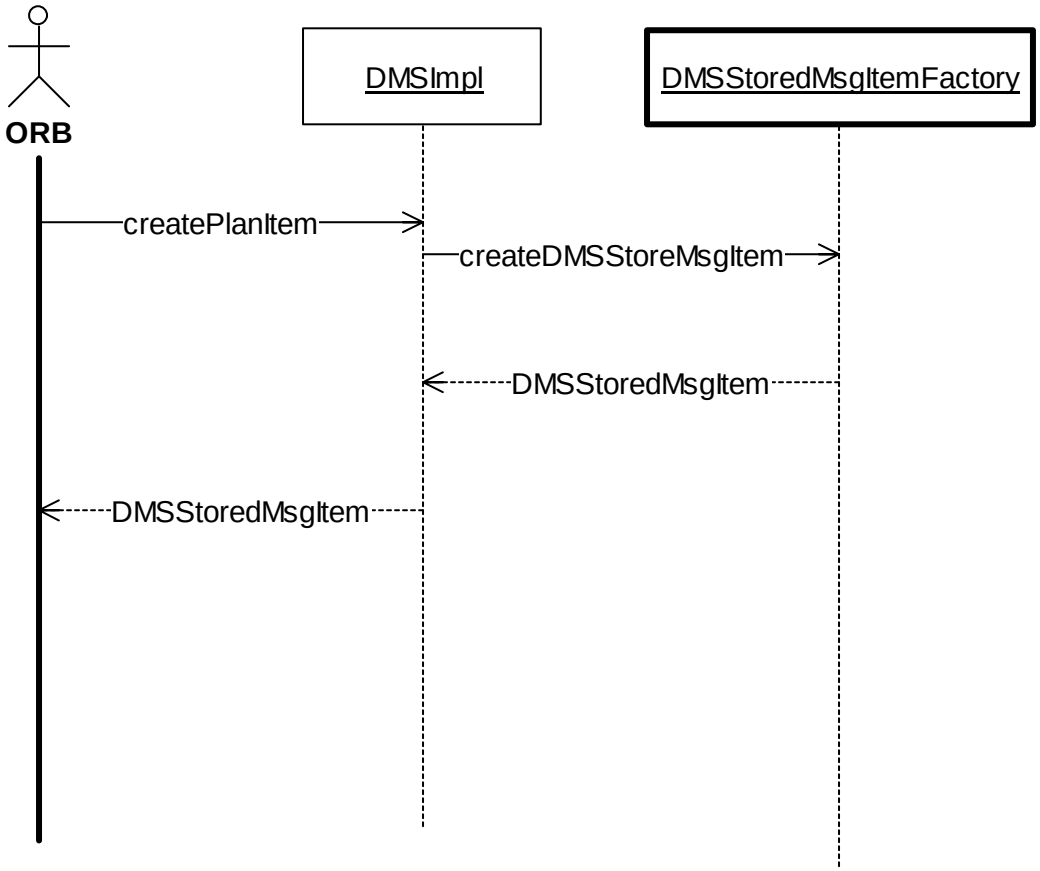

Figure 3-19. DMSControlModule:CreatePlanItem (Sequence Diagram)

3.2.3.15 DMSControlModule:SetMessage (Sequence Diagram)

This sequence diagram shows the required operations to allow a message to be set on a DMS. Since field communications are involved, the actual setting of the message is performed asynchronously via a command queue. Prior to queing the command, preliminary checks are done to ensure the request can be completed given operational rules. A Command Status object is used by the caller to track the progress of the asynchronous command. Events are pushed on an event channel after the message is changed and when the controlling operations center has been changed.

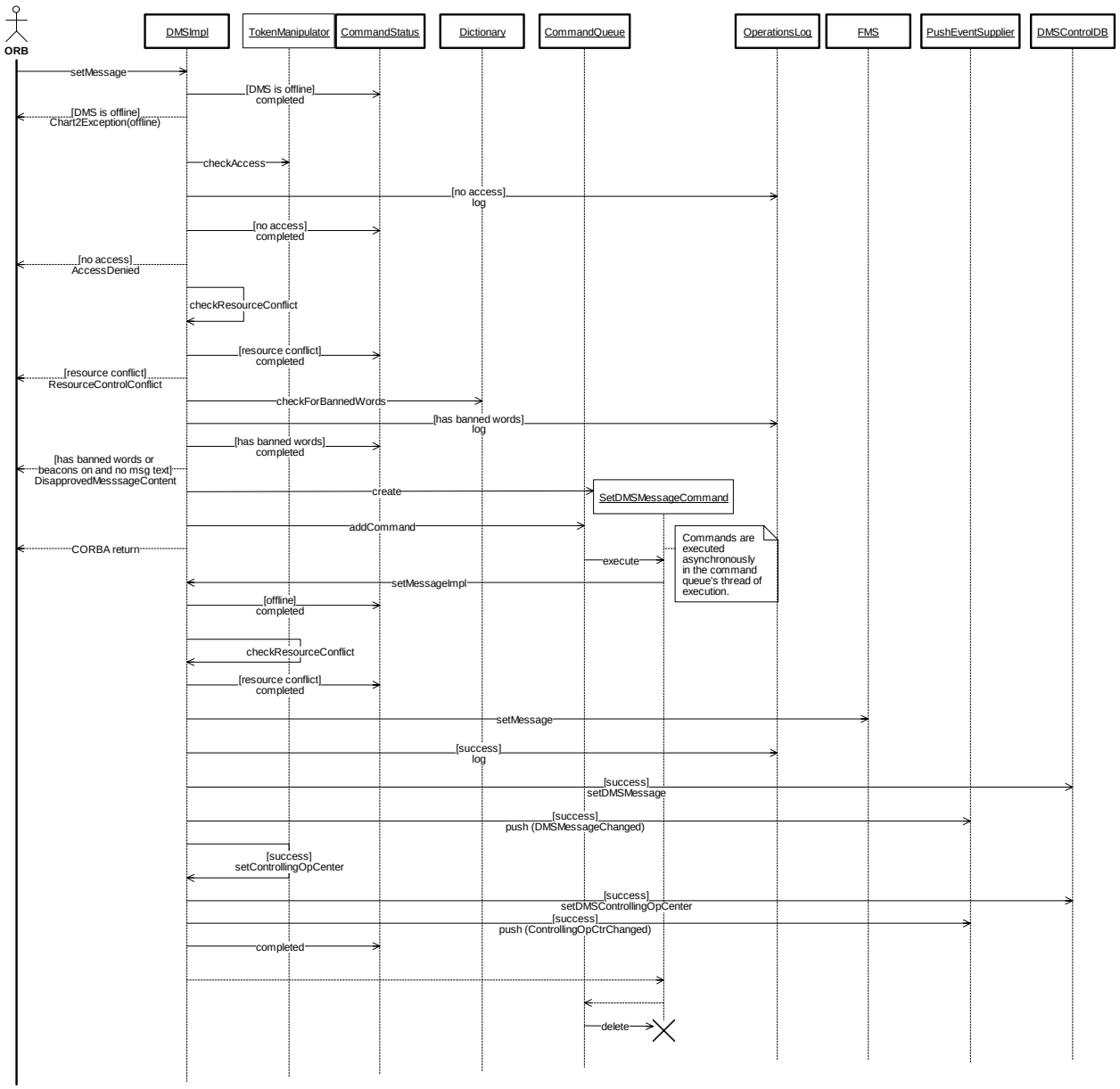

Figure 3-20. DMSControlModule:SetMessage (Sequence Diagram)

3.2.3.16 DMSControlModule:PollDMS (Sequence Diagram)

A user with the proper functional rights can poll a DMS for its current status outside of the normal polling cycle. Since this will require field communications which may be time consuming, the command is executed asynchronously by the DMS and a command status object is used to keep the caller apprised of the execution status. An event is pushed via the CORBA Event Service to notify the caller and others of the new DMS status following the poll attempt.

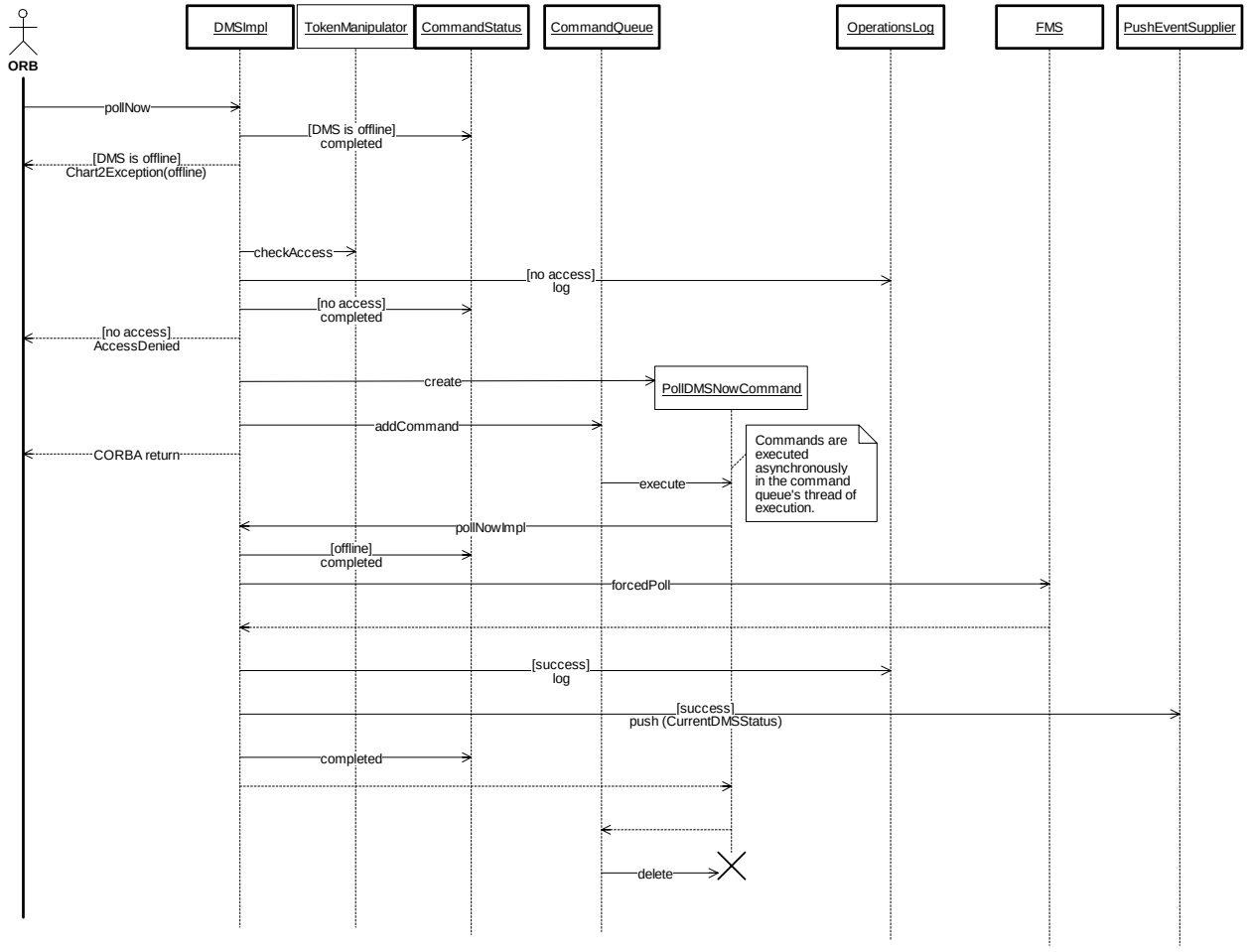

Figure 3-21. DMSControlModule:PollDMS (Sequence Diagram)

3.2.3.17 DMSControlModule:ResetController (Sequence Diagram)

A user with the proper functional rights can reset a DMS controller. Since this will require field communications which may be time consuming, the command is executed asynchronously by the DMS and a command status object is used to keep the caller apprised of the execution status. The DMS is blanked prior to the reset command being issued to ensure that the DMS will be reset to a known state. An event is pushed via the CORBA Event Service to notify the caller and others that the sign was blanked. Since resetting a DMS relinquishes control of the DMS, resource conflict must be checked to make sure the DMS is not in use by different operations center than the one requesting the reset.

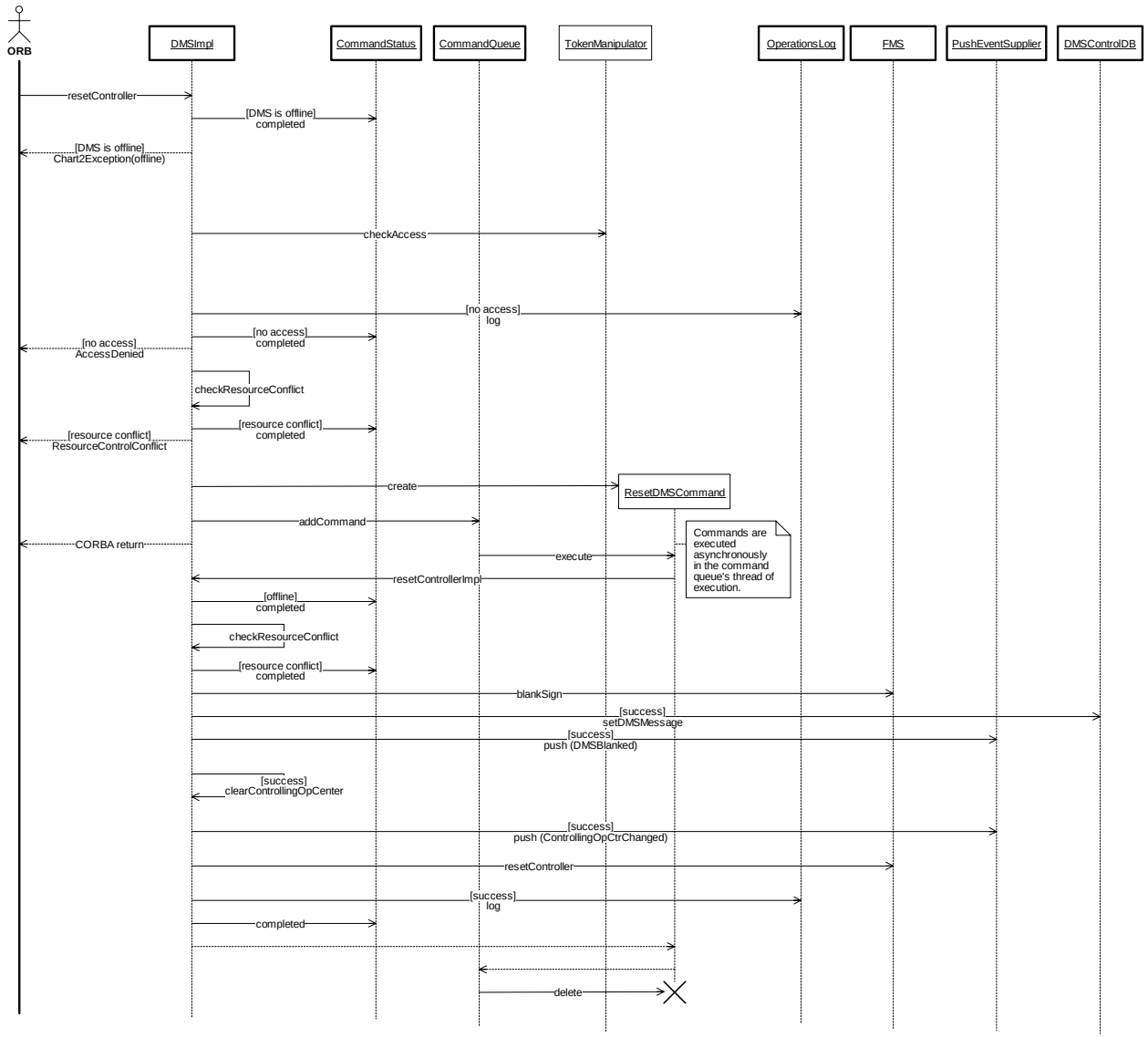

Figure 3-22. DMSControlModule:ResetController (Sequence Diagram)

3.2.3.18 DMSControlModule:SetCommLossTimeout (Sequence Diagram)

When the commLossTimeout parameter for a DMS is to be set, a SetCommLossTimeoutCommand object is created and added to the command queue. The caller who is setting the commLossTimeout is informed about the command being queued and is required to determine the status of the set operation from the CommandStatus object. When the command added to the CommandQueue gets its turn to execute, it calls back into the DMSImpl, which sends the request to the FMS and updates the database. After execution the setCommLossTimeoutCommand object is deleted.

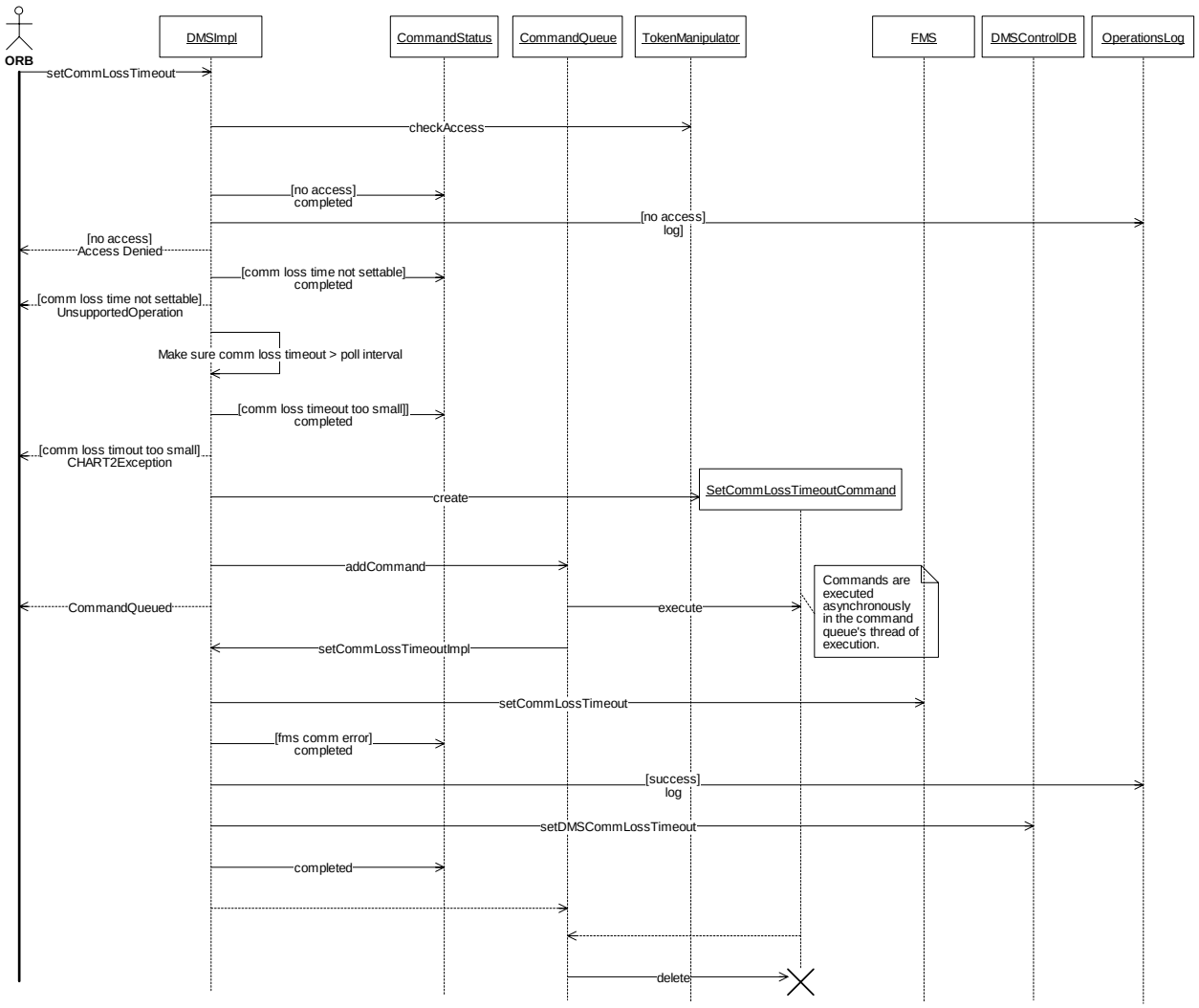

Figure 3-23. DMSControlModule:SetCommLossTimeout (Sequence Diagram)

3.2.3.19 DMSControlModule:SetDMSOffline (Sequence Diagram)

A user with the proper functional rights can set a DMS offline if the DMS is blank or failed. Taking a DMS offline involves FMS communications and may take an extended amount of time. For this reason, the operation is executed asynchronously and a command status object is used to keep the caller informed of the execution status. An attempt is made to blank the DMS before taking it offline. Taking the DMS offline has the effect of stopping automatic polling and disallows any further operations other than to put the DMS online. Shared resource management rules apply to this operation. If the DMS is under the control of an operations center, only a user from that operations center or a user with override functional rights may take the DMS offline. Taking the DMS offline clears the controlling operations center.

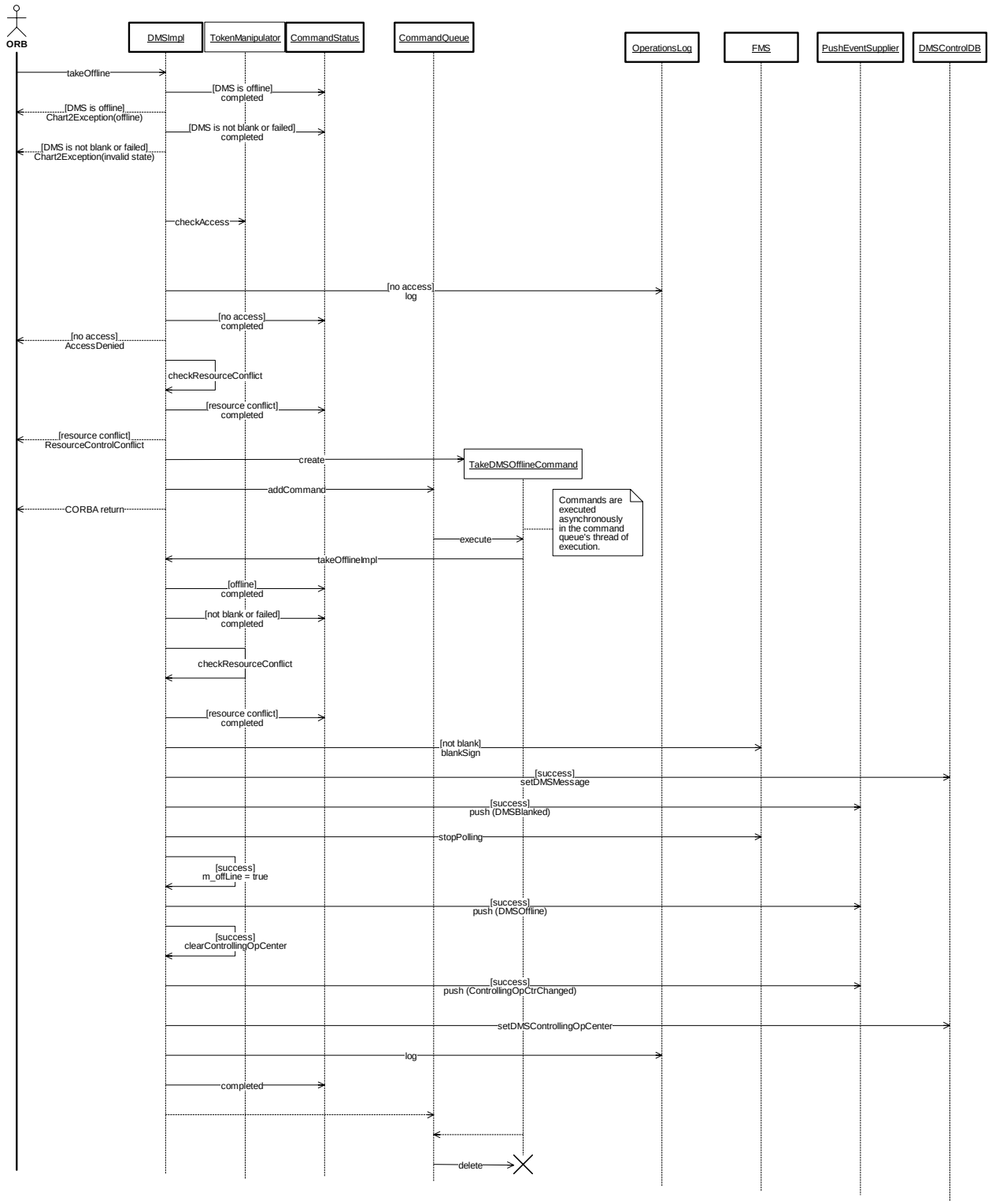

Figure 3-24. DMSControlModule:SetDMSOffline (Sequence Diagram)

3.2.3.20 DMSControlModule:SetDMSOnline (Sequence Diagram)

A user with the proper functional rights may put a DMS online. Putting the DMS online involves FMS communications and is therefore done asynchronously. A command status object is used by the caller to monitor the status of the operation. Before a DMS is brought online, it is blanked to insure its status is consistent with the status known by the system. Automatic polling of the DMS is started within the FMS subsystem.

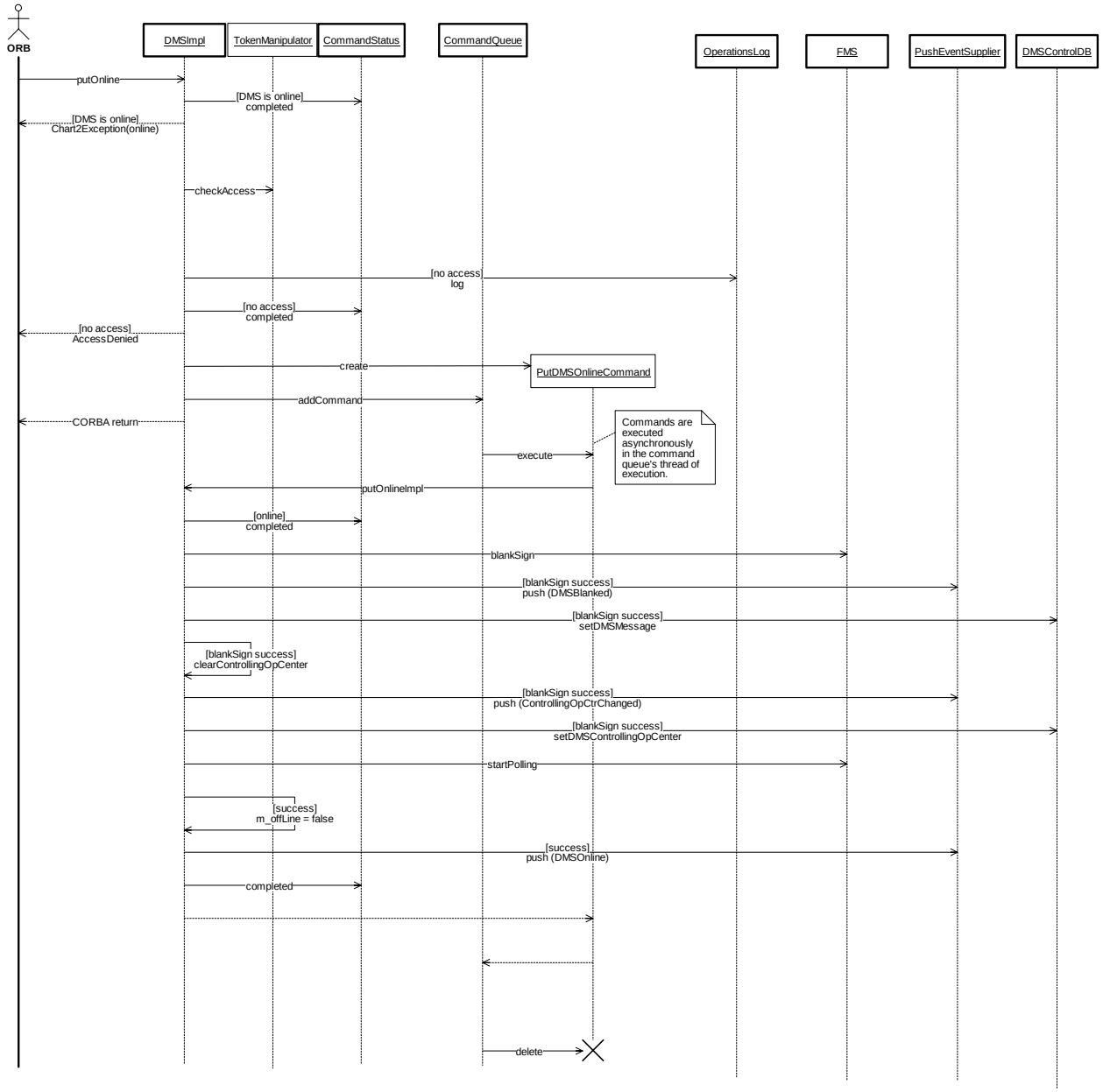

Figure 3-25. DMSControlModule:SetDMSOnline (Sequence Diagram)

3.2.3.21 DMSControlModule:SetPollInterval (Sequence Diagram)

When the pollInterval parameter for a DMS is set, a SetPollIntervalCommand object is created and added to the command queue. The caller who is setting the pollInterval is informed about the command being queued and is required to determine the status of the set operation from the CommandStatus object. When the command added to the CommandQueue gets its turn to execute, the poll interval request is sent to the FMS and the database is updated. After execution the SetPollIntervalCommand object is deleted.

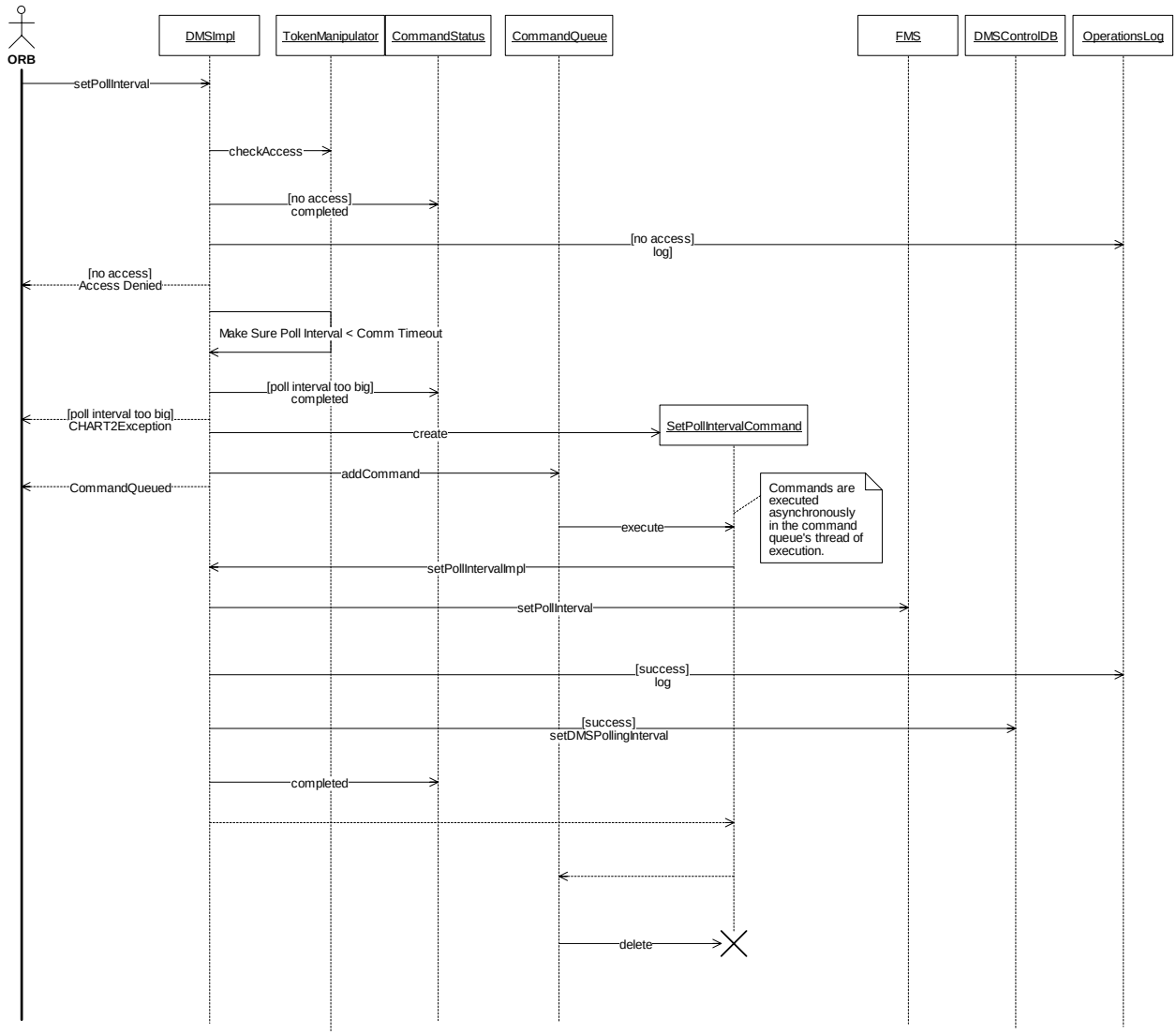

Figure 3-26. DMSControlModule:SetPollInterval (Sequence Diagram)

3.2.3.22 DMSControlModule:Shutdown (Sequence Diagram)

The DMSControlModule is shutdown by its host application. When told to shutdown, the DMSControlModule disconnects the DMSFactory from the ORB, withdraws its offer from the trader, and shuts down the object. When the DMSFactory is shut down, it withdraws the offers of each DMS and disconnects each DMS from the ORB. The DMS Control module also disconnects any DMSStoredMsgItem objects that it is serving.

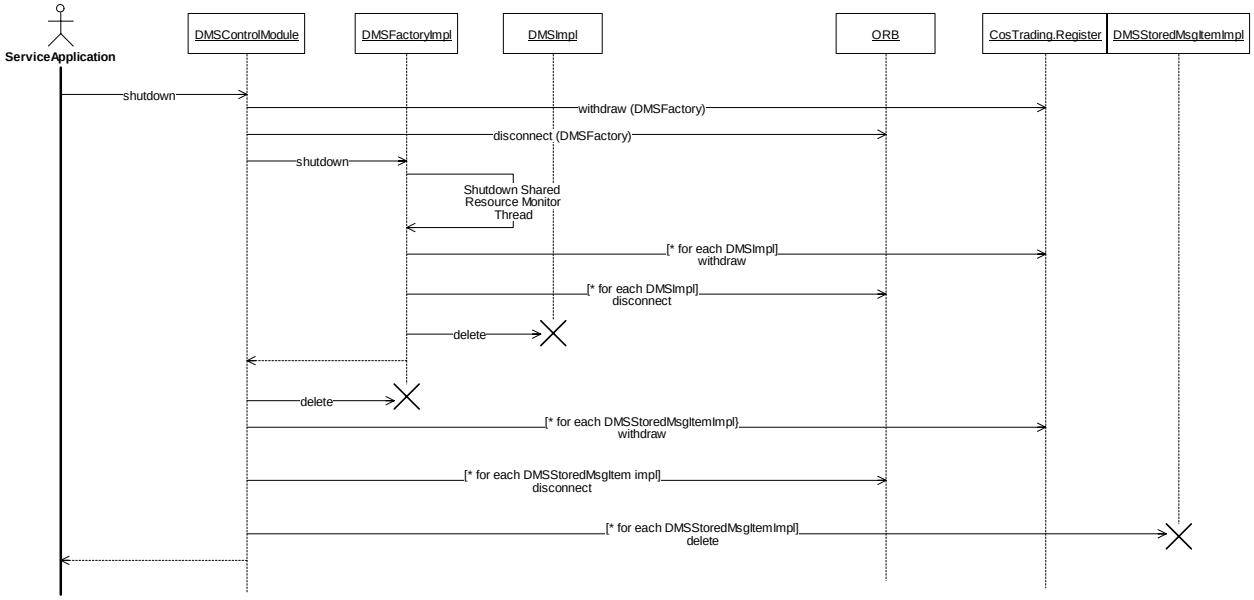

Figure 3-27. DMSControlModule:Shutdown (Sequence Diagram)

3.3 DMSLibraryModule

3.3.1 DMSMessageLibraryClasses (Class Diagram)

The DMSLibraryModule is a Service Application module that serves the DMSLibraryFactory, DMSMessageLibrary and StoredDMSMessage objects to the rest of the Chart2 system.

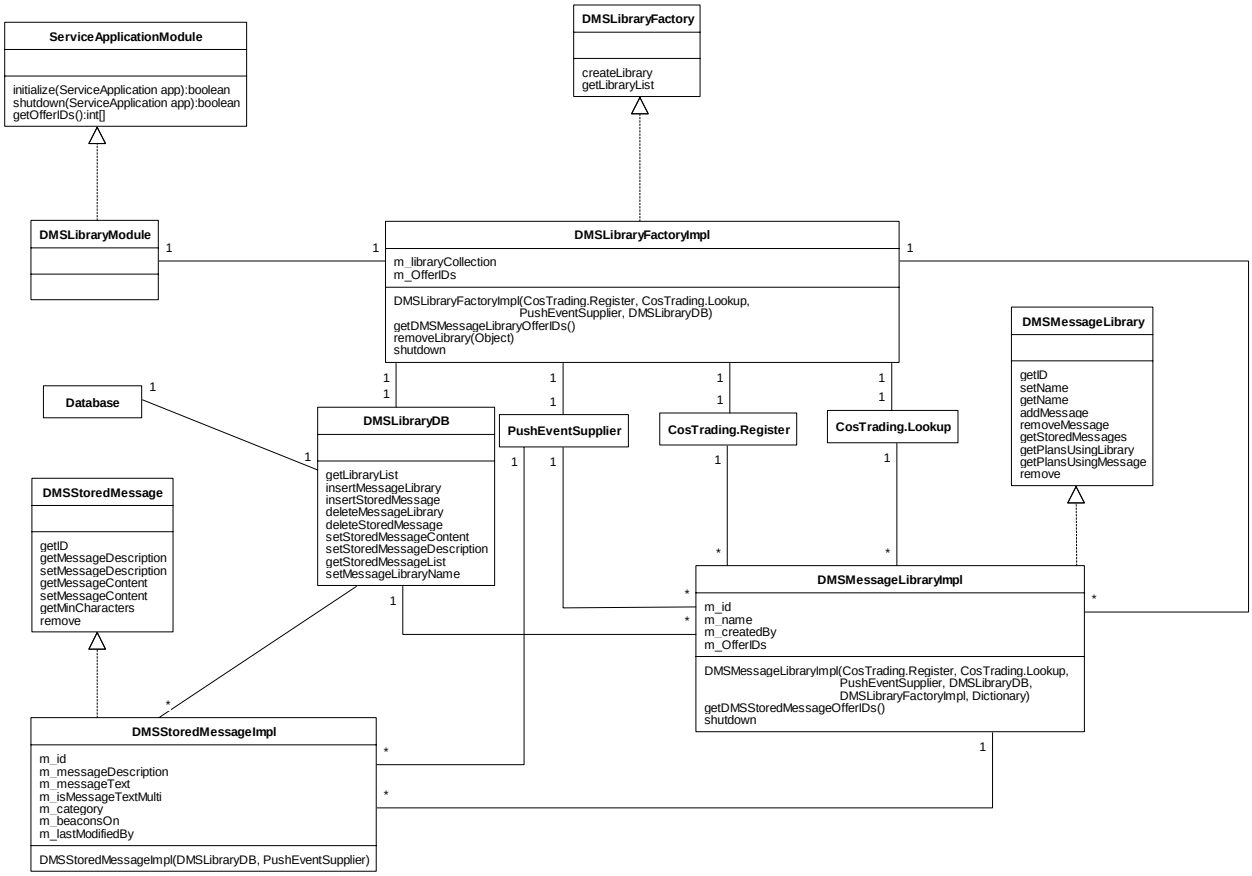

Figure 3-28. DMSMessageLibraryClasses (Class Diagram)

3.3.1.1 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

1

interface

3.3.1.2 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

1

interface

3.3.1.3 Database (Class)

1

3.3.1.4 DMSLibraryDB (Class)

This class contains the methods that perform database operations for the DMS Library module. It is constructed with a Database object that provides the connections to the database server. All the methods in this class get a unused connection from the database before performing any operation on the database. The connection is released at completion of the operation.

3.3.1.5 DMSLibraryFactory (Class)

This class is used to create new DMS libraries and maintain them in a collection.

interface

3.3.1.6 DMSLibraryFactoryImpl (Class)

This class implements the DMSLibraryFactory interface as specified in the IDL from the High Level Design and is used to create new DMS message libraries and manage them in a collection

3.3.1.7 DMSMessageLibrary (Class)

This class represents a logical collection of stored DMS messages that are stored in the database.

interface

3.3.1.8 DMSMessageLibraryImpl (Class)

This class implements the DMSMessageLibrary interface as specified in the IDL from the High Level Design. It represents a logical collection of DMS messages which are stored in the database.

3.3.1.9 DMSStoredMessage (Class)

This class represents a stored DMS message which is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

interface

3.3.1.10 DMSStoredMessageImpl (Class)

This class implements the DMSStoredMessage interface as specified in the IDL from the High Level Design. It represents a DMS stored message which is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

3.3.1.11 DMSLibraryModule (Class)

This module manages the Message Libraries and Stored Messages for the DMS service. It provides the functionality to add, delete and modify the libraries and messages stored in them.

3.3.1.12 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

1

3.3.1.13 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.3.2 Sequence Diagrams

3.3.2.1 DMSLibraryModule:CreateDMSMessageLibrary (Sequence Diagram)

This sequence diagram shows how a user possessing the proper functional rights can add a DMS Message Library to the system. An AccessDenied exception is returned if the user does not have the functional right to add a library to the system. Otherwise, the library object is created and published via the CORBA Trading Register. An event is pushed via the CORBA Event Service to notify interested parties of the new library. The library is added to the database. The user action is logged to the operations log.

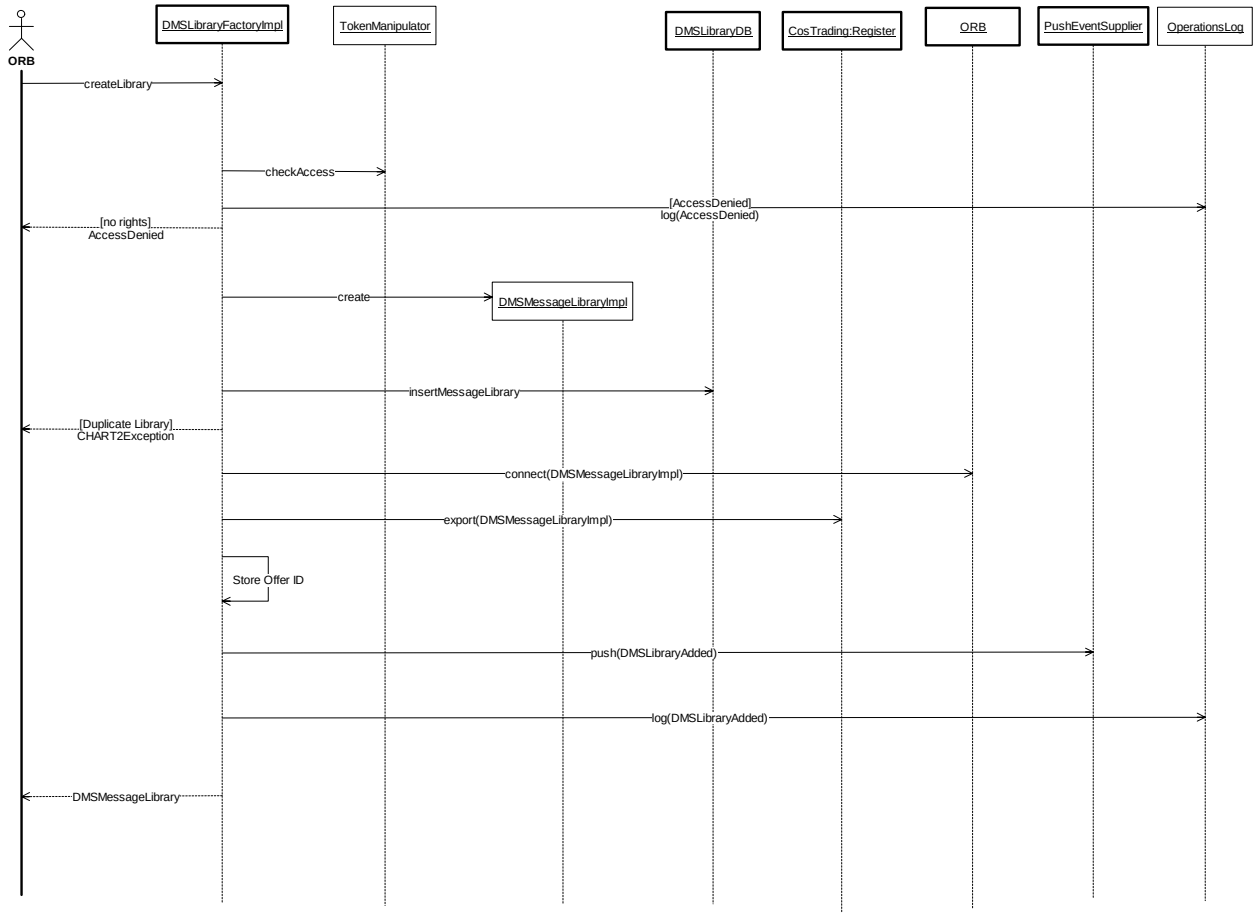

Figure 3-29. DMSLibraryModule:CreateDMSMessageLibrary (Sequence Diagram)

3.3.2.2 DMSLibraryModule:CreateDMSStoredMessage (Sequence Diagram)

This sequence diagram shows how a user with the proper functional rights can create a new DMS message to be stored for later use. An AccessDenied exception is returned if a user without proper functional rights tries to add a message to a library. The contents of the message are checked against a dictionary prior to storing. The dictionary is obtained by querying the trader. If approved, the message is stored in the database, an object is created, connected to the ORB, published in the trader and its existence is pushed to interested parties via the CORBA Event Service. Note that even though a dictionary check is done at the time of storage, the dictionary is always checked on the server side prior to allowing a message to be set on a DMS. The user action is logged to the operations log.

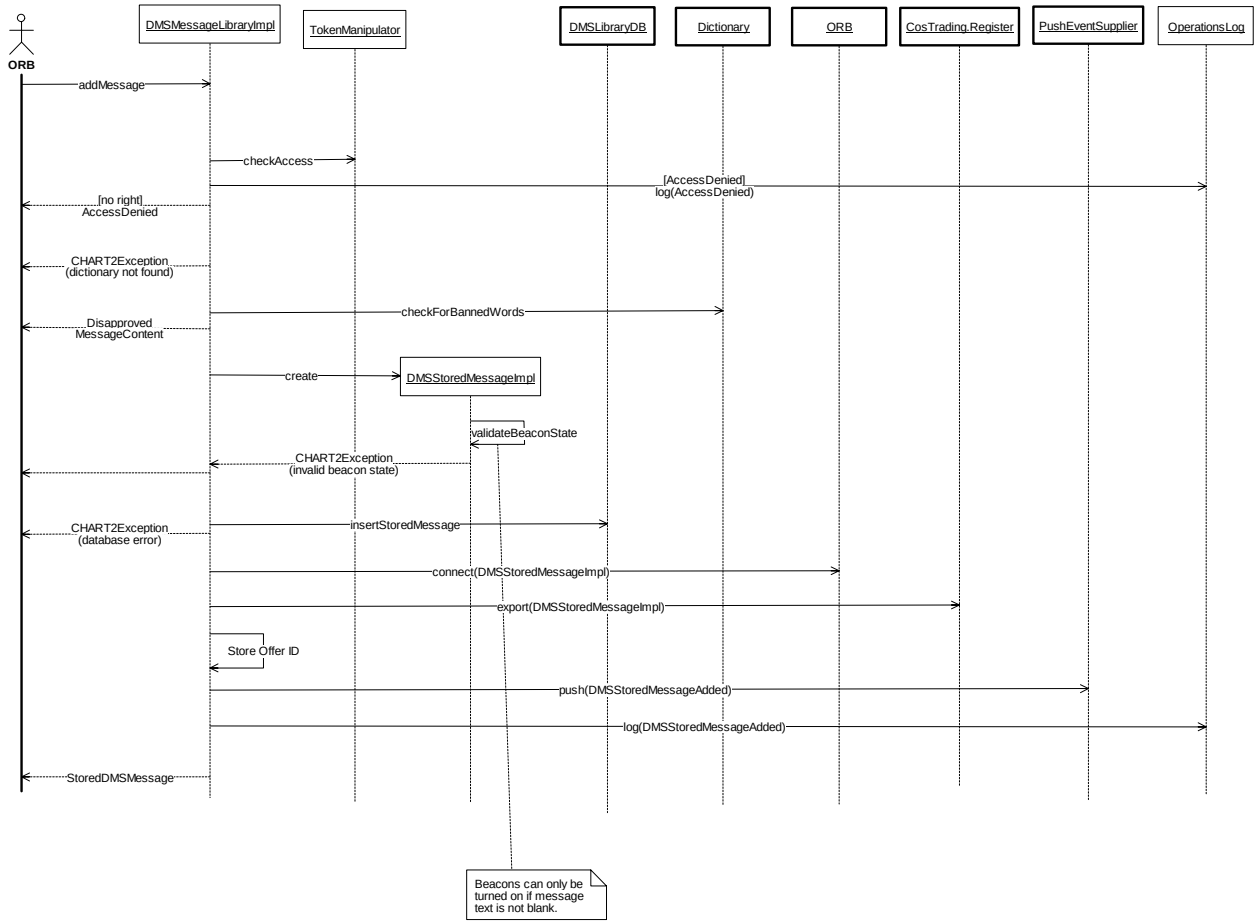

Figure 3-30. DMSLibraryModule:CreateDMSStoredMessage (Sequence Diagram)

3.3.2.3 DMSLibraryModule:GetPlansUsingLibrary (Sequence Diagram)

This sequence diagram shows how a user can get a list of plans that are using the stored DMS messages of a particular DMS message library.

Figure 3-31. DMSLibraryModule:GetPlansUsingLibrary (Sequence Diagram)

3.3.2.4 DMSLibraryModule:GetPlansUsingMessage (Sequence Diagram)

This sequence diagram shows how a user can get a list of plans that are using a particular stored DMS message.

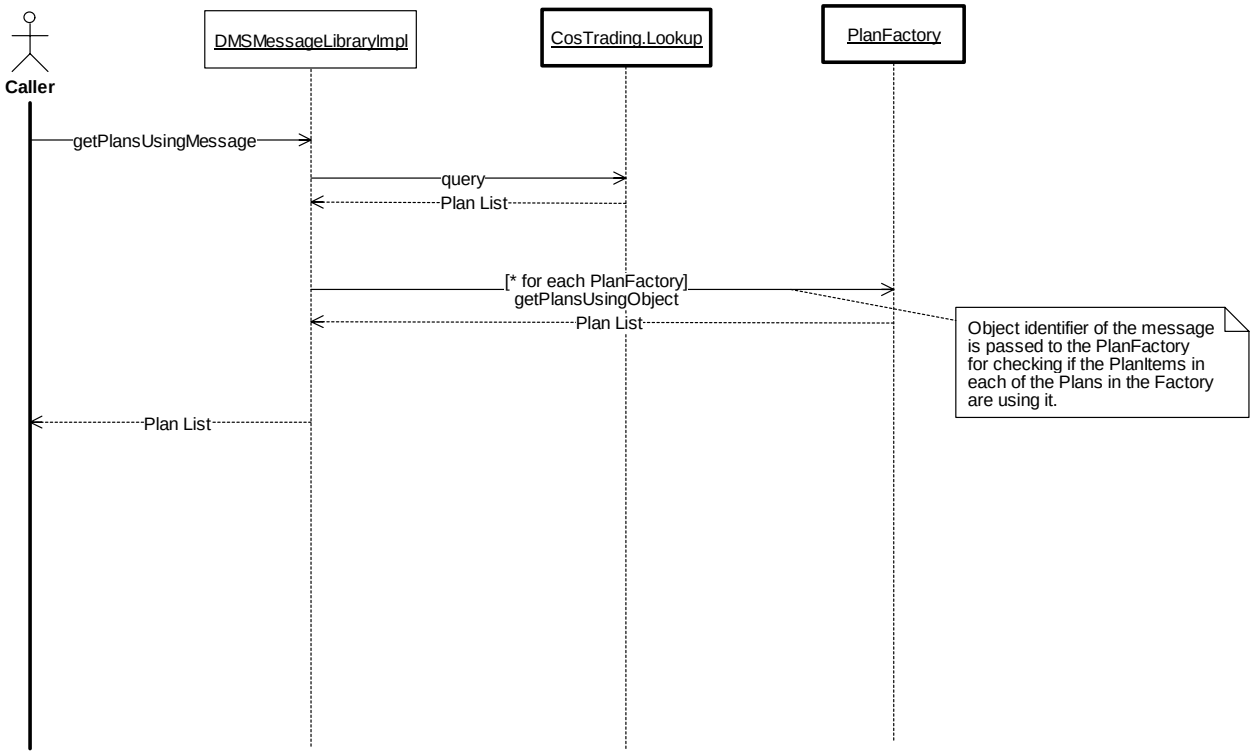

Figure 3-32. DMSLibraryModule:GetPlansUsingMessage (Sequence Diagram)

3.3.2.5 DMSLibraryModule:Initialize (Sequence Diagram)

This sequence diagram shows the startup for the DMS Library Module. This module will be created by a ServiceApplication, which provides the access to the basic services needed by this module such as ORB, Trader and Database. This module creates the DMS Library specific database object. It also creates a DMS Library Factory and DMS Message Library objects contained in the Library Factory. The DMSLibraryFactory and DMSMessageLibrary objects are connected to the ORB and published in the CORBA Trading service to make them available to other processes. The Offer IDs of the objects that were published in the trader are stored in a file so that they may be wihdrawn at shutdown. This module creates an event supplier channel for pushing events to other processes and publishes it in the trader.

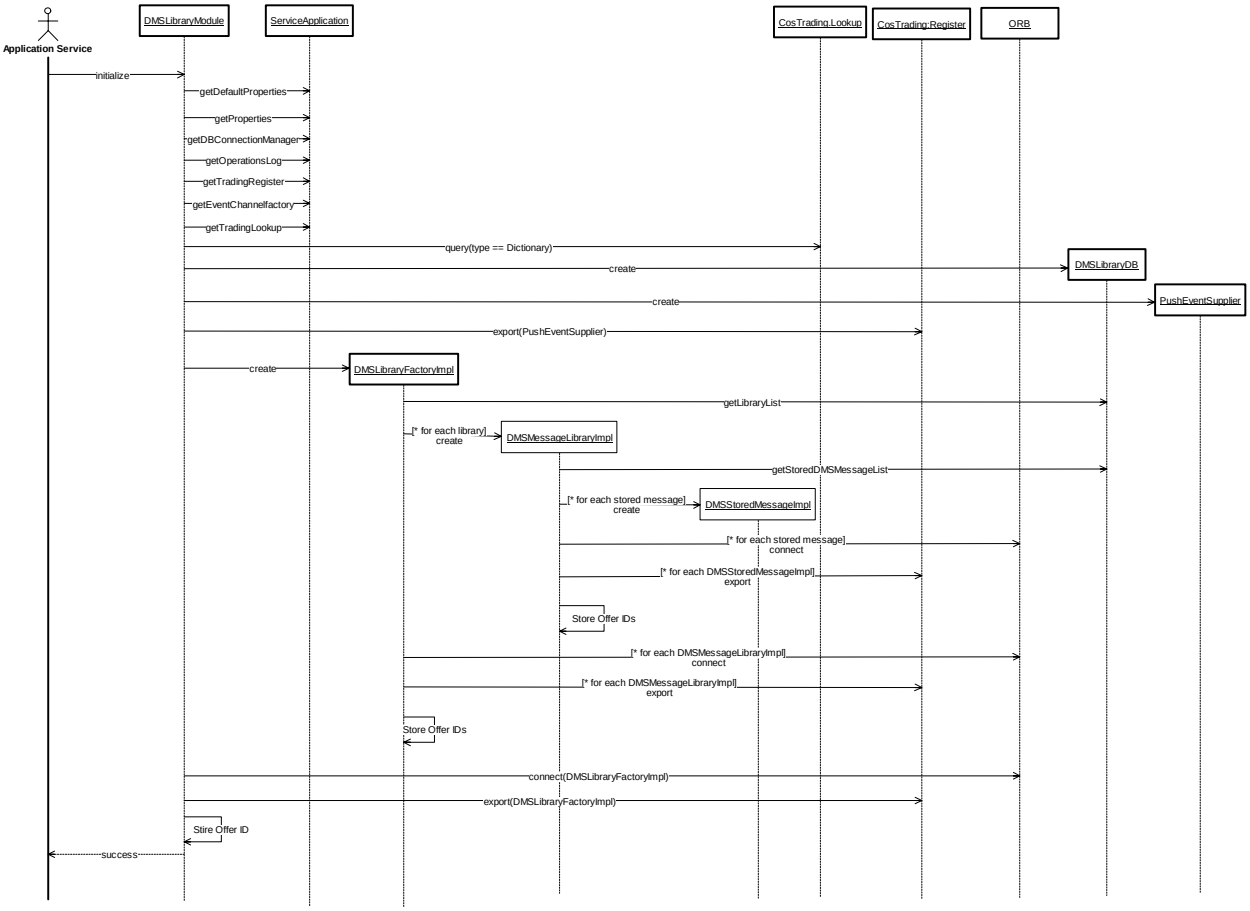

Figure 3-33. DMSLibraryModule:Initialize (Sequence Diagram)

3.3.2.6 DMSLibraryModule:ModifyDMSStoredMessage (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can modify a stored message in a library. An AccessDenied exception is returned if the user does not have the functional right to modify the message. Otherwise, the message passed is checked against the dictionary for banned words and a disapproved message content exception is returned if the message contains banned words. The beacon state is checked to make sure that the beacons are not set for a blank message. An event is pushed via the CORBA Event Service to notify others of the change to the stored message's contents. The user action are logged to the operation log

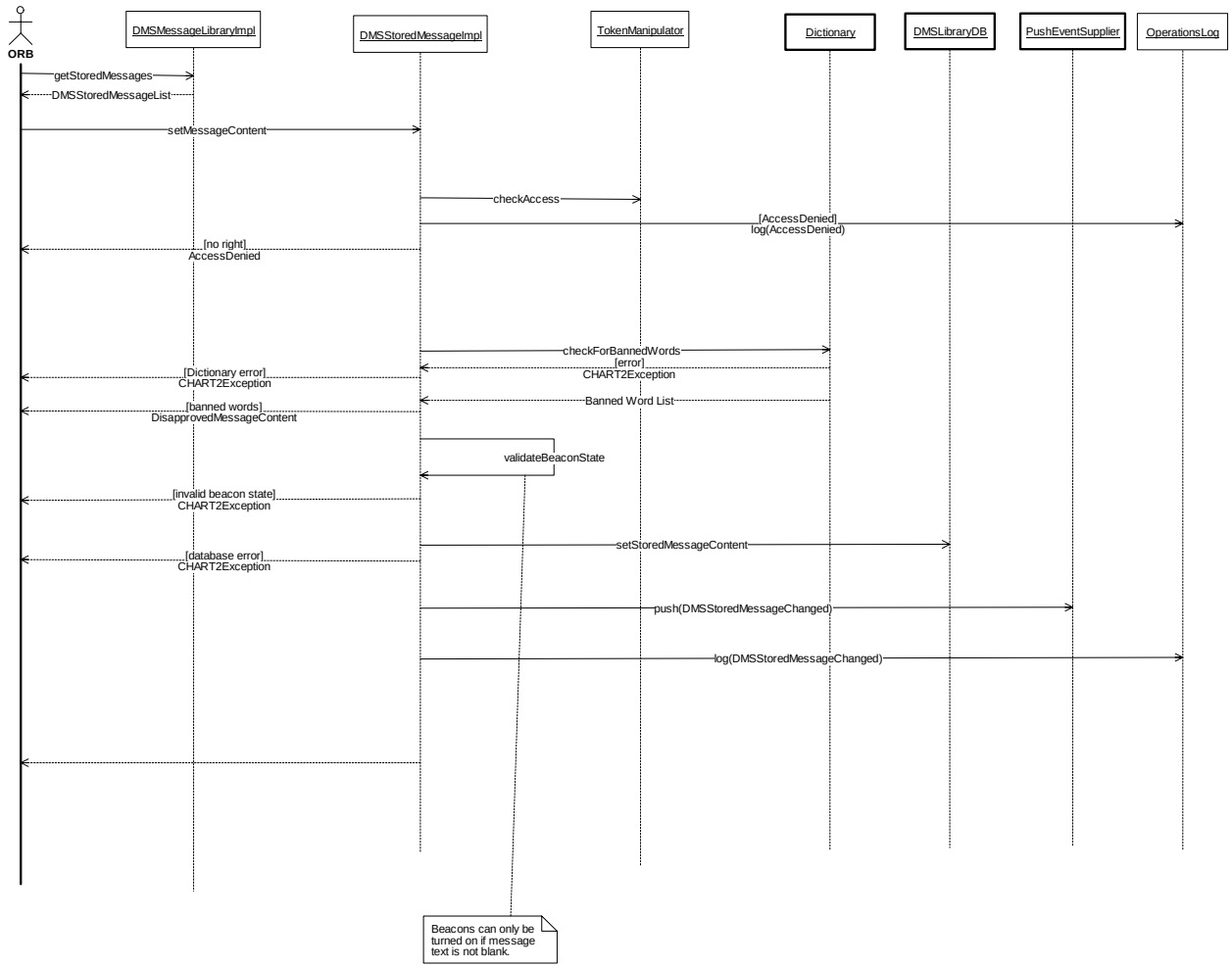

Figure 3-34. DMSLibraryModule:ModifyDMSStoredMessage (Sequence Diagram)

3.3.2.7 DMSLibraryModule:RemoveDMSMessageLibrary (Sequence Diagram)

This sequence diagram shows how a user with the proper functional rights can remove a DMS Message Library from the system. This will include the removal of all stored messages contained within the library. Since stored messages may be used in Plans that contain DMSStoredMessageItems, a check is made for any plans that may contain the stored messages being deleted and the user is warned. If the user acknowledges the deletions, each message within the library is removed from the trader and the system, events are pushed to notify others of the action, and the library is removed from the Trading Service. An AccessDenied exception is returned if the user does not have functional rights to delete a library. The library and the stored messages are also deleted from the database. User actions are logged into the operations log.

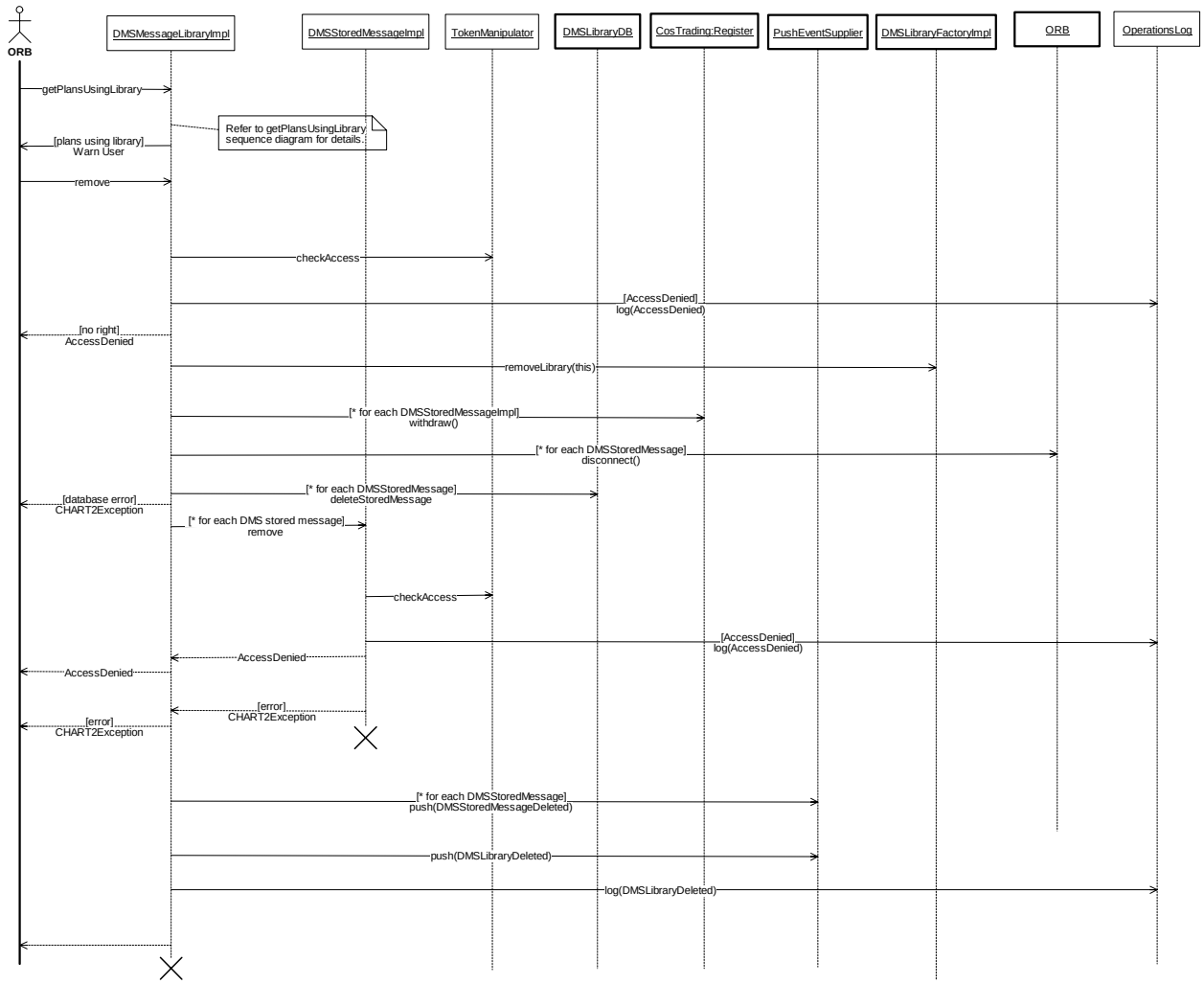

Figure 3-35. DMSLibraryModule:RemoveDMSMessageLibrary (Sequence Diagram)

3.3.2.8 DMSLibraryModule:RemoveDMSMessageLibraryFromFactory (Sequence Diagram)

This sequence diagram shows how a DMS Message Library object is removed from the Library Factory when a Message Library is deleted from the system.

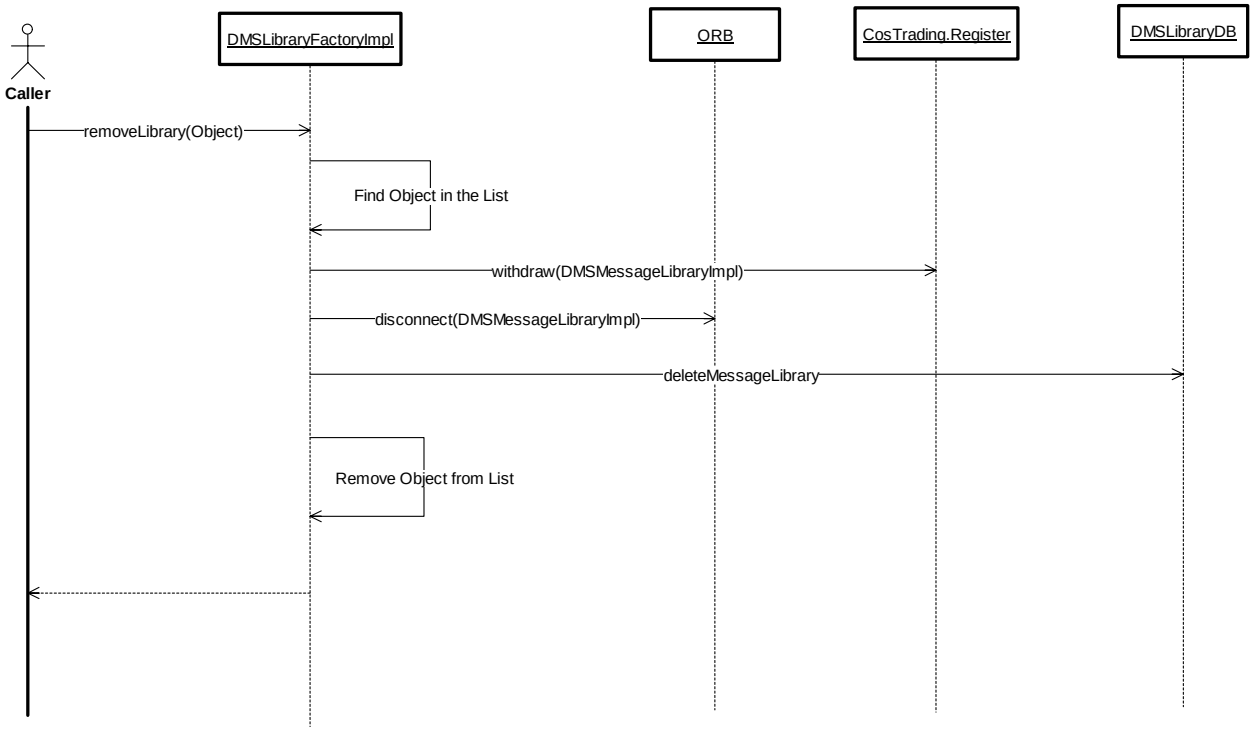

Figure 3-36. DMSLibraryModule:RemoveDMSMessageLibraryFromFactory (Sequence Diagram)

3.3.2.9 DMSLibraryModule:RemoveDMSStoredMessage (Sequence Diagram)

This sequence diagram shows how a user with the proper functional rights may remove a stored DMS message from the system. Since a stored DMS message may be used in a plan, a check is made to see if the message is used in a plan so that the user can be warned accordingly. The act of deleting the stored message involves withdrawing the object from the trader, disconnecting it from the ORB, updating the database, deleting the object and pushing an event to notify others that the message has been removed from its library. An AccessDenied exception is returned if the user does not have the functional right to delete a stored message. The user action is logged into the operations log.

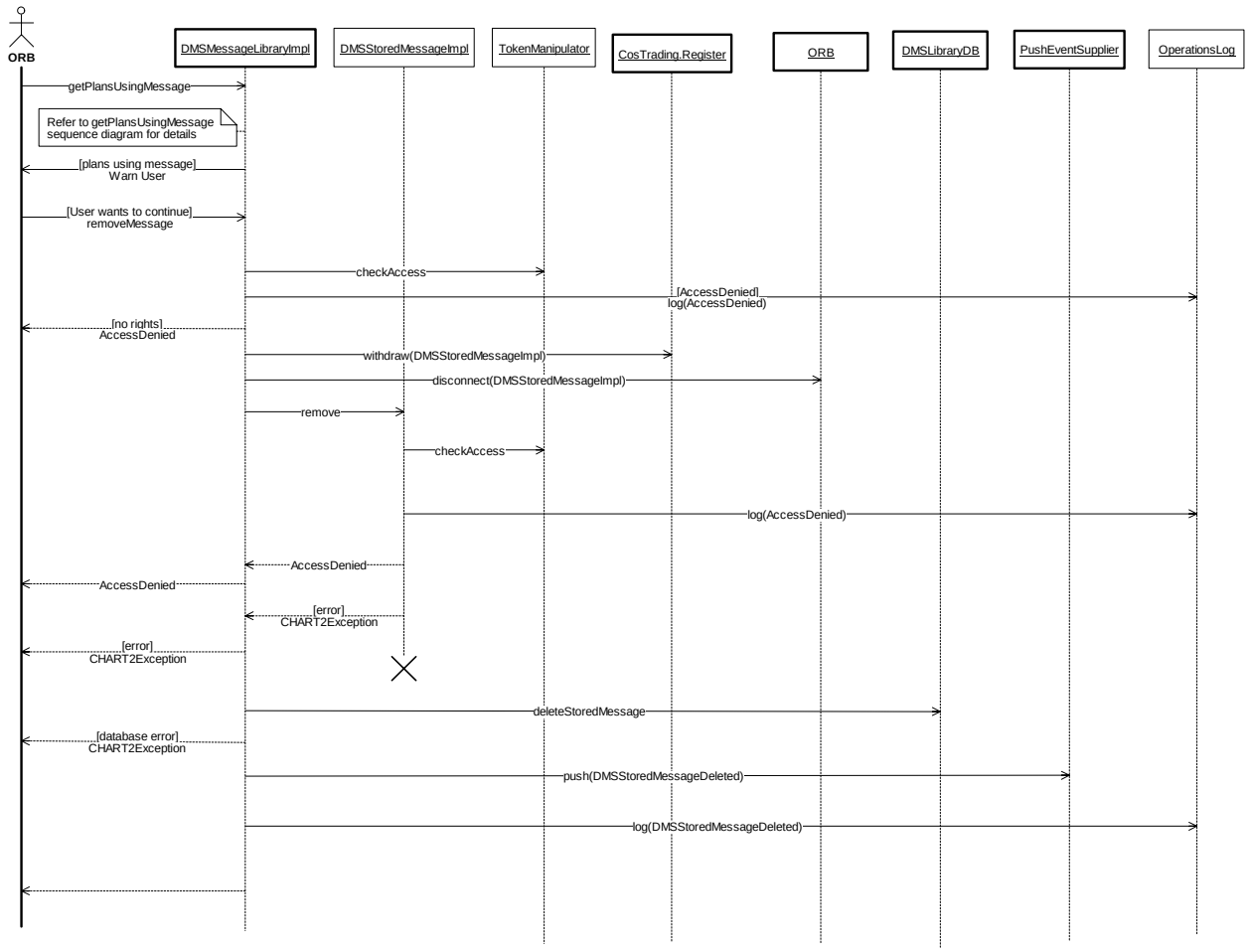

Figure 3-37. DMSLibraryModule:RemoveDMSStoredMessage (Sequence Diagram)

3.3.2.10 DMSLibraryModule:SetDMSMessageLibraryName (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can set the name of a message library. An AccessDenied exception is returned if the user does not have the functional right to set the library name. Otherwise, the database is updated and an event is pushed via the CORBA event service to notify others of the new library name. The user action is logged to the operations log.

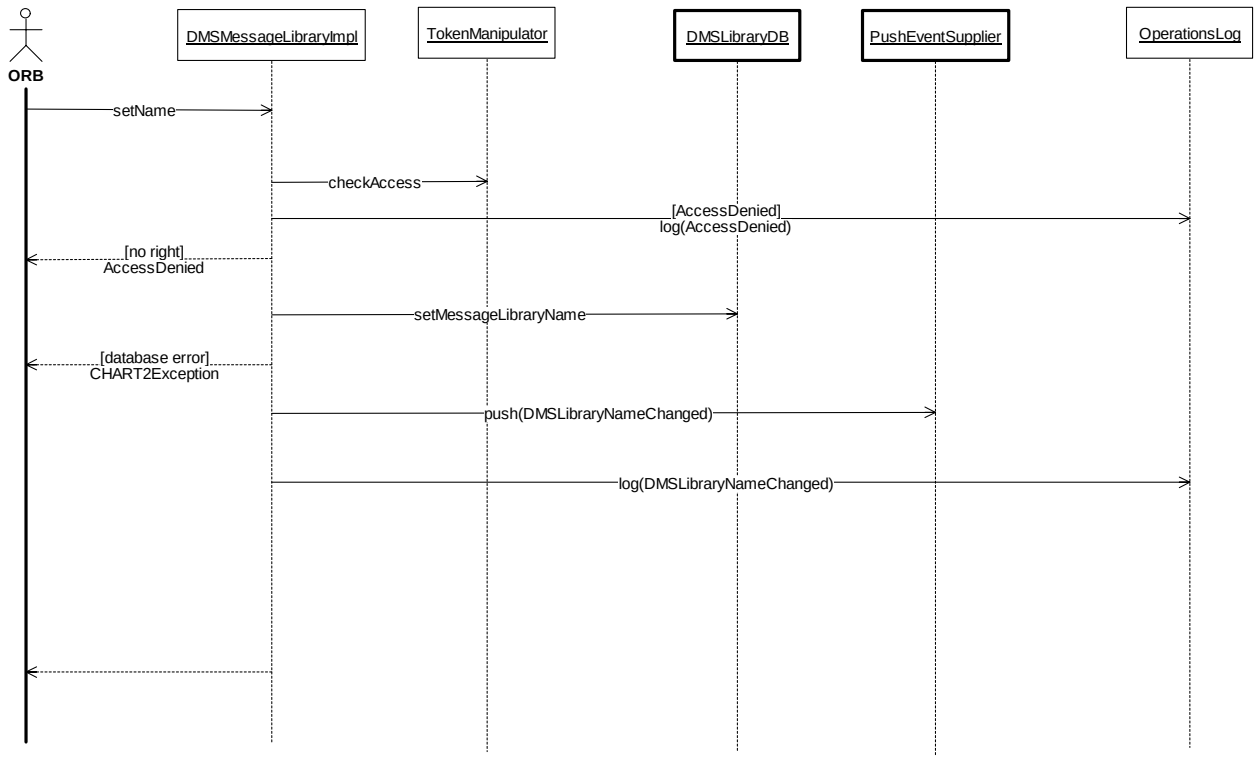

Figure 3-38. DMSLibraryModule:SetDMSMessageLibraryName (Sequence Diagram)

3.3.2.11 DMSLibraryModule:SetDMSStoredMessageName (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can set the name of a Stored DMS message. An AccessDenied exception is returned if the user does not have the functional right to set the library name. Otherwise, the database is updated. The user action is logged to the operations log.

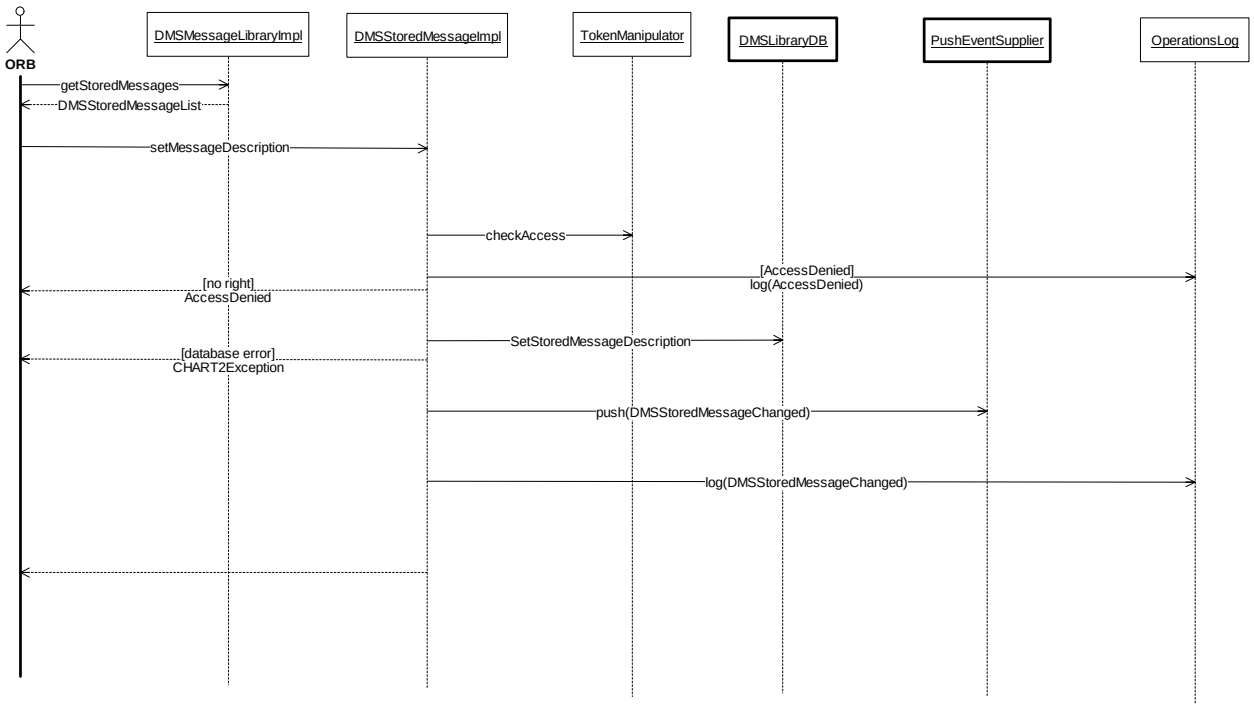

Figure 3-39. DMSLibraryModule:SetDMSStoredMessageName (Sequence Diagram)

3.3.2.12 DMSLibraryModule:Shutdown (Sequence Diagram)

This sequence diagram shows how the Message library module is shut down by the service application. This module will withdraw all the offers that were published in the Trader and delete the objects that were created by it.

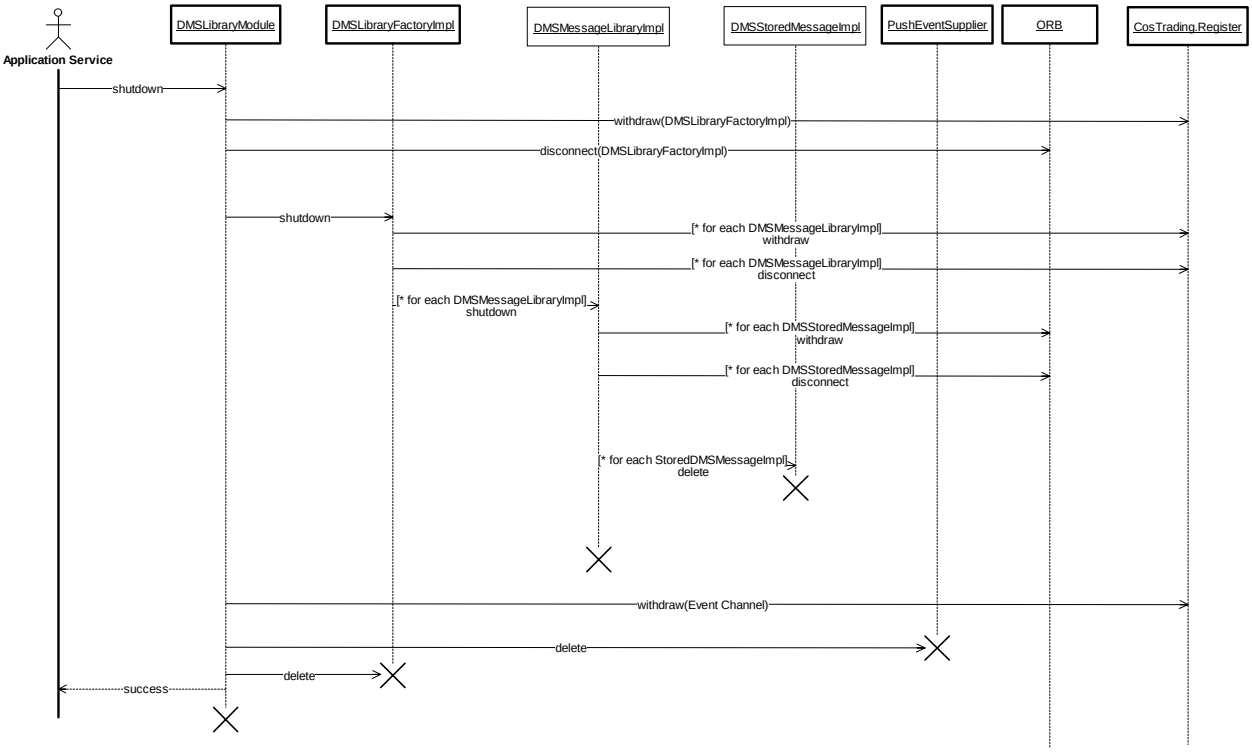

Figure 3-40. DMSLibraryModule:Shutdown (Sequence Diagram)

3.4 DictionaryModule

3.4.1 DictionaryModClassDiagram (Class Diagram)

The DictionaryModule is a Service Application module that creates and serves the Dictionary implementation to the rest of the Chart2 system.

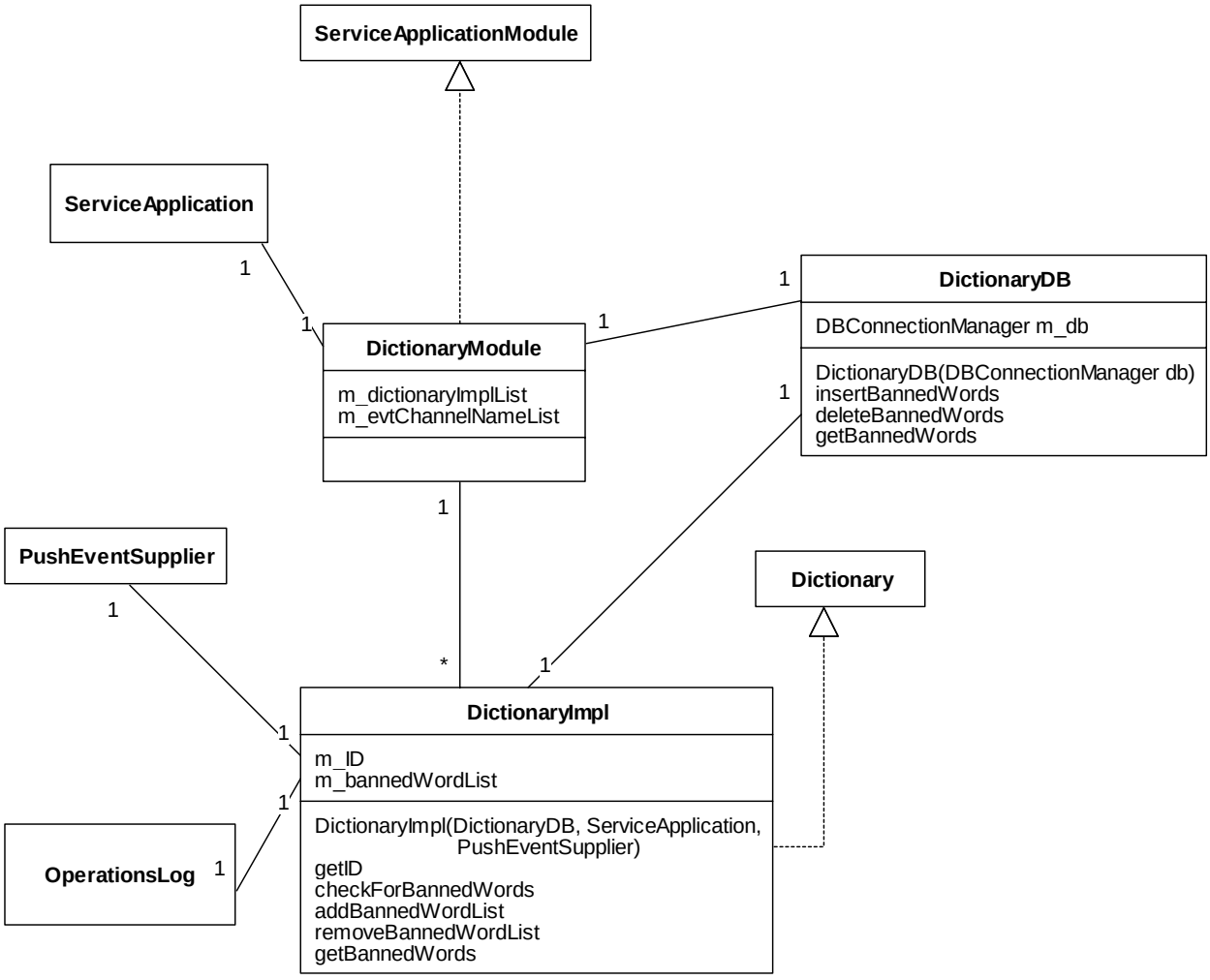

Figure 3-41. DictionaryModClassDiagram (Class Diagram)

3.4.1.1 Dictionary (Class)

This class is used to check for banned words in a message that may be displayed on a DMS. In addition to methods for checking the words, it has methods to allow the contents of the dictionary to be changed.

1

interface

3.4.1.2 DictionaryImpl (Class)

This class implements the Dictionary as specified by the IDL. It provides functionality to add, delete and check for words that are banned from being used in a DMS message

3.4.1.3 DictionaryModule (Class)

This class implements the Service Application module interface. It publishes the dictionary implementation.

3.4.1.4 DictionaryDB (Class)

This class provides API calls to add, remove and retrieve banned words from the database. The connection to the database is acquired from the Database object which manages all the database connections.

3.4.1.5 OperationsLog (Class)

This class provides the functionality to add a log entry to the Chart II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

1

3.4.1.6 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

1

3.4.1.7 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

1

interface

3.4.1.8 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

1

interface

3.4.2 Sequence Diagrams

3.4.2.1 DictionaryImpl:addBannedWordList (Sequence Diagram)

The given list of words is added to the banned words dictionary database and the copy of the dictionary in memory is also updated. The newly added banned words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the "Manage Dictionary" privilege.

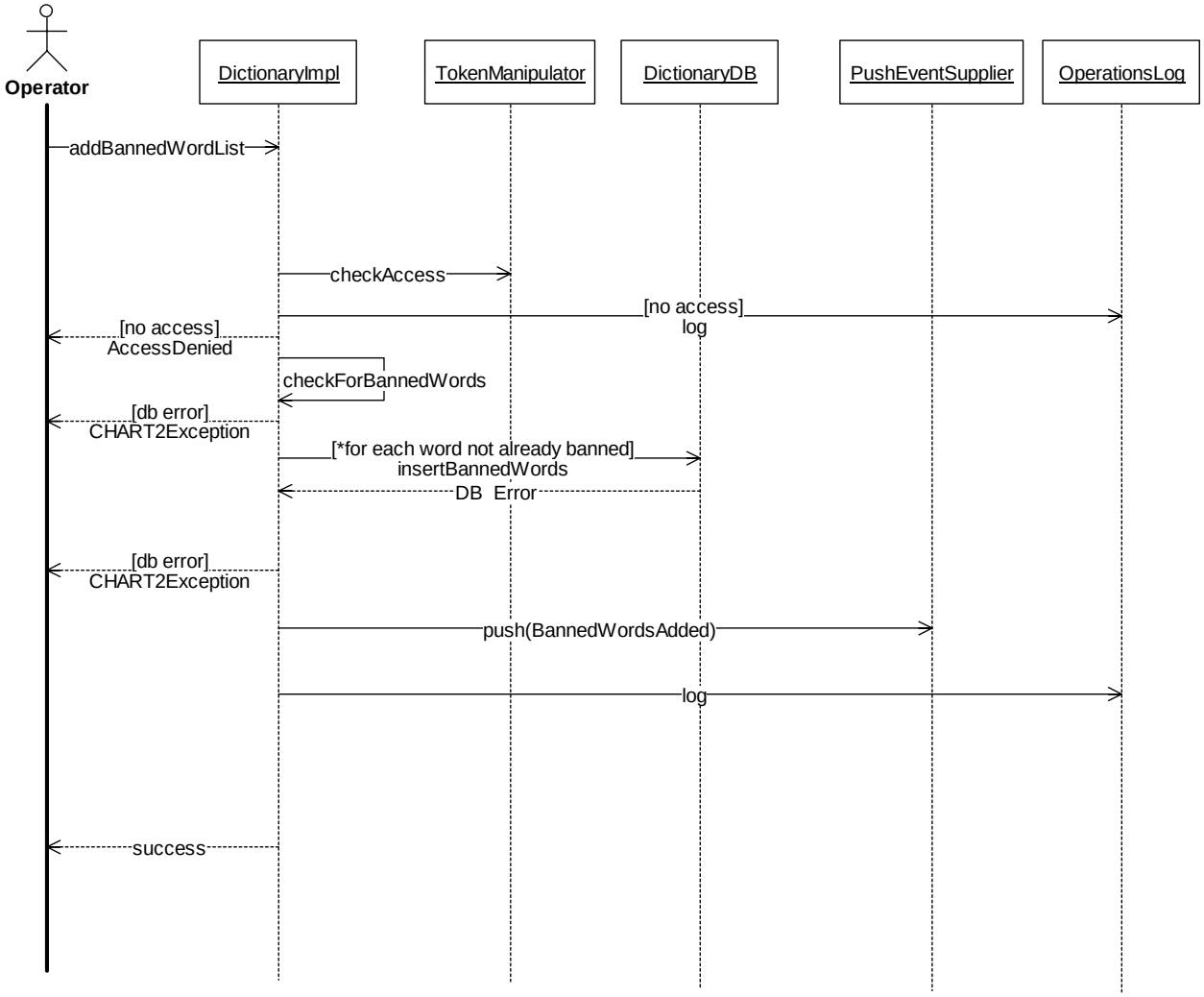

Figure 3-42. DictionaryImpl:addBannedWordList (Sequence Diagram)

3.4.2.2 DictionaryImpl:checkForBannedWords (Sequence Diagram)

The string provided by the operator is scanned for any banned words by looking up the database. Any character from the given set of delimiters is taken to be a valid delimiter of words in the string. The list of banned words present in the string is returned.

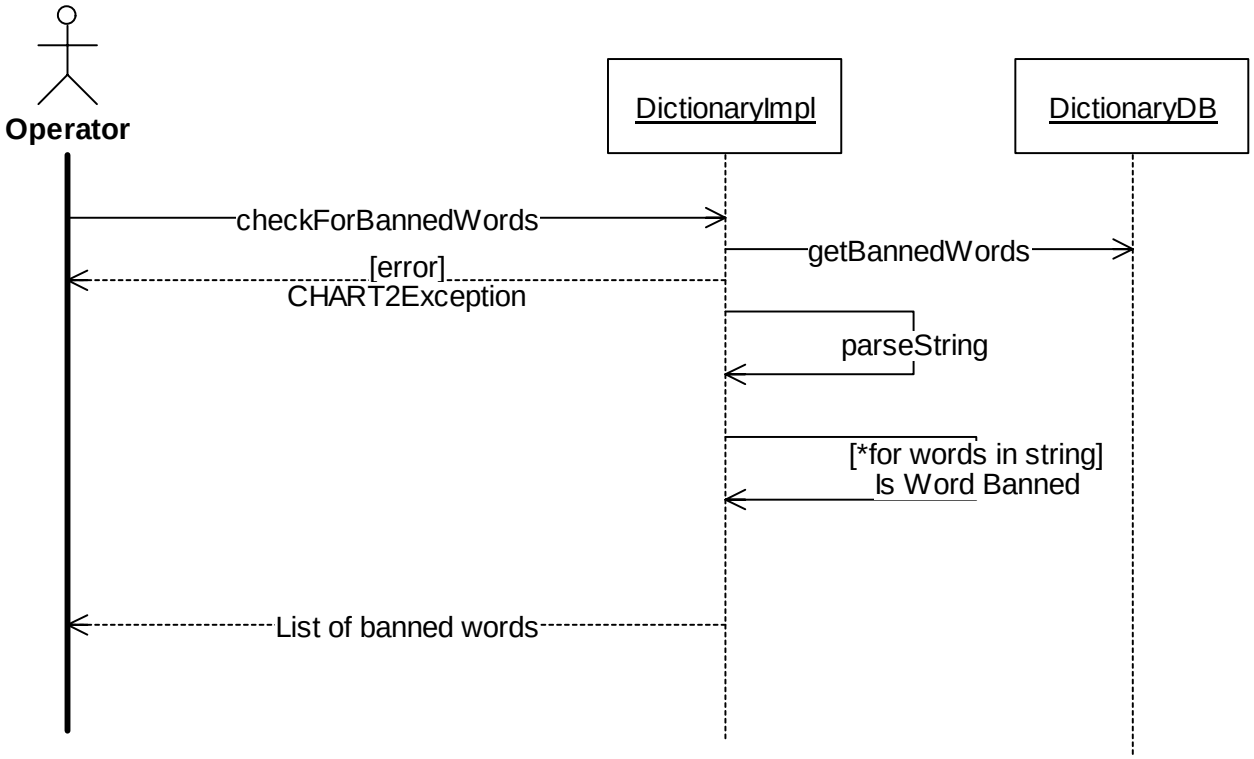

Figure 3-43. DictionaryImpl:checkForBannedWords (Sequence Diagram)

3.4.2.3 DictionaryImpl:getBannedWords (Sequence Diagram)

The list of banned words in the dictionary is read from the database and returned to the operator. Access is denied to any operator without the “Manage Dictionary” privilege.

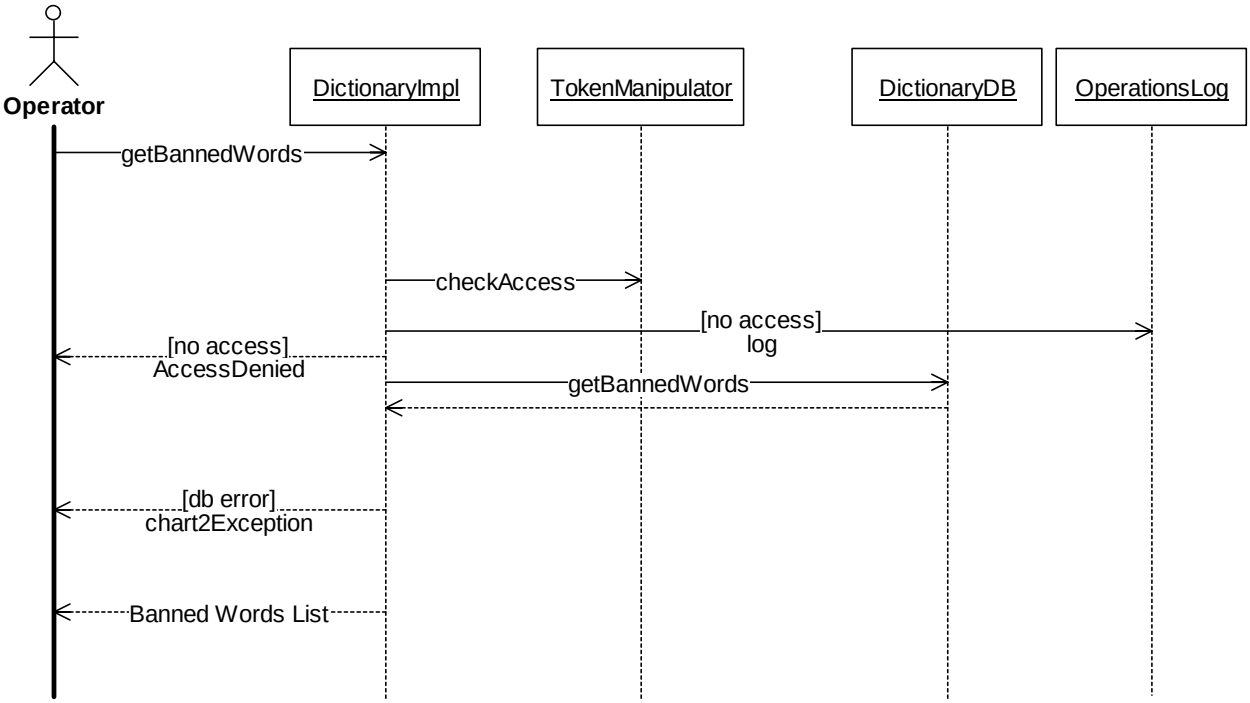

Figure 3-44. DictionaryImpl:getBannedWords (Sequence Diagram)

3.4.2.4 DictionaryImpl:removeBannedWordList (Sequence Diagram)

The given list of words is removed from the banned words dictionary database. The removed words are then communicated to the dictionary event consumers by invoking the push operation. Access is denied to any operator without the "Manage Dictionary" privilege.

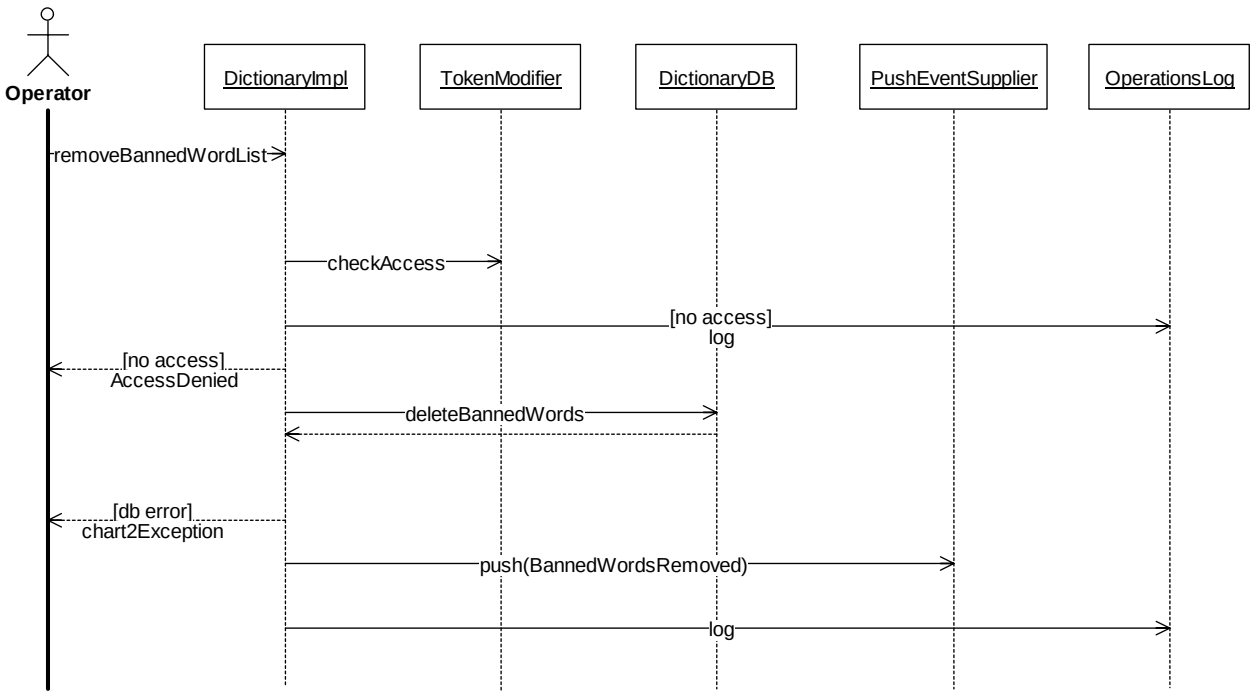

Figure 3-45. DictionaryImpl:removeBannedWordList (Sequence Diagram)

3.4.2.5 DictionaryModule:initialize (Sequence Diagram)

When the DMS service calls the initialize method of Dictionary module, the dictionary objects are created, connected to the ORB, exported to the CORBA trading service. The dictionary objects are now available to serve the consumers.

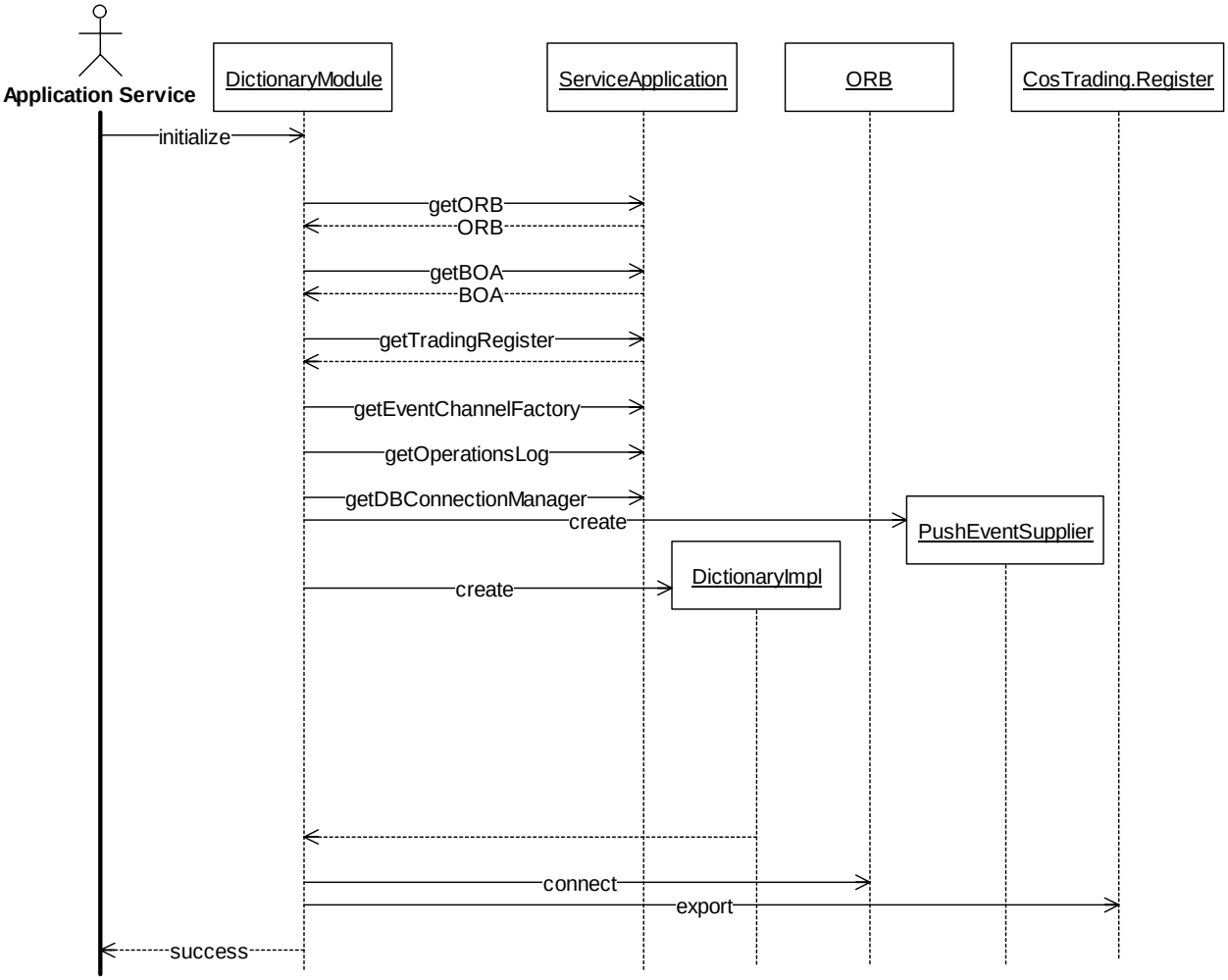

Figure 3-46. DictionaryModule:initialize (Sequence Diagram)

3.4.2.6 DictionaryModule:shutdown (Sequence Diagram)

When the host service application calls shutdown in the Dictionary module, the dictionary object is withdrawn from the CORBA trading service and disconnected from the ORB. The objects are then deleted.

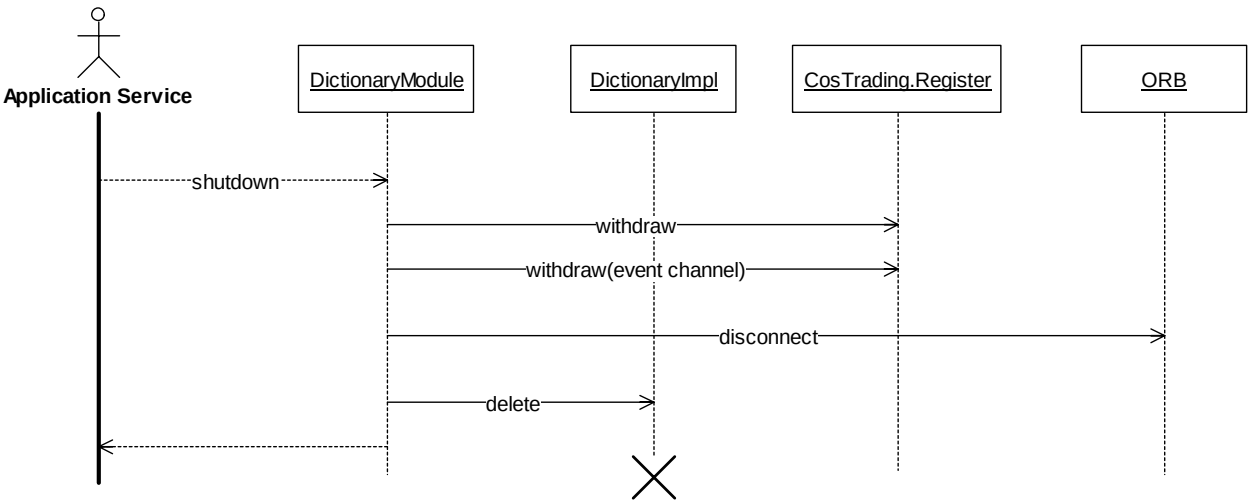

Figure 3-47. DictionaryModule:shutdown (Sequence Diagram)

3.5 PlanService

3.5.1 PlanServiceClasses (Class Diagram)

The PlanService is an application that helps in installation and termination of the modules related to Plan service.

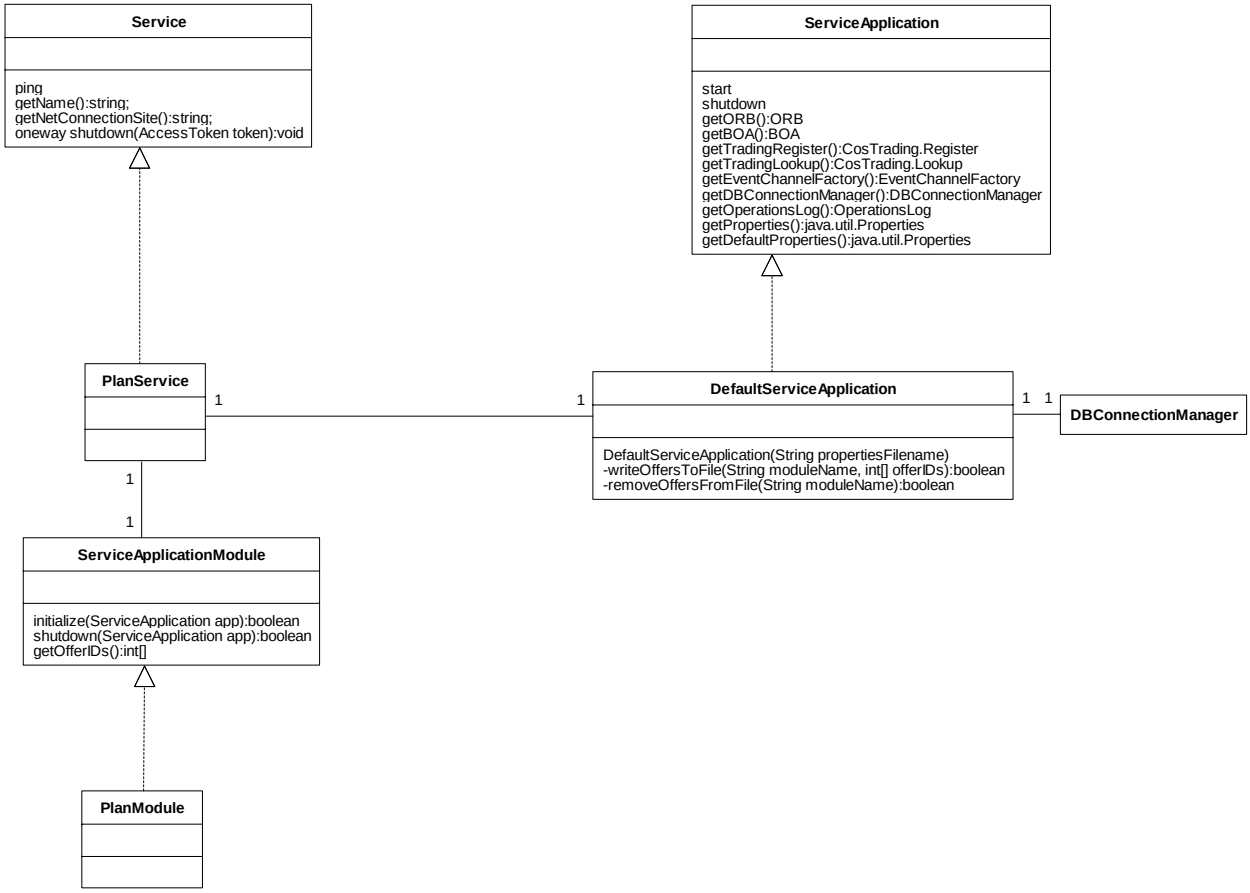

Figure 3-48. PlanServiceClasses (Class Diagram)

3.5.1.1 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.5.1.2 PlanModule (Class)

This module creates, publishes and deletes the object that implements the PlanFactory interface.

3.5.1.3 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

1

3.5.1.4 PlanService (Class)

This class provides the main method for the Plan Service Application. It initializes and shuts down the PlanModule. It makes use of the DefaultServiceApplication to provide access to standard objects to the server modules.

3.5.1.5 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

interface

3.5.1.6 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.5.1.7 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.5.2 Sequence Diagrams

3.5.2.1 PlanService:Shutdown (Sequence Diagram)

This diagram shows the sequence of operations that are performed when a Plan Service is shutdown. PlanService sends a shutdown message to the PlanModules that are open and destroys them. It also calls the shutdown method of DefaultServiceApplication. Refer to the DefaultServiceApplcation's shutdown sequence diagram in the Utility package for datails.

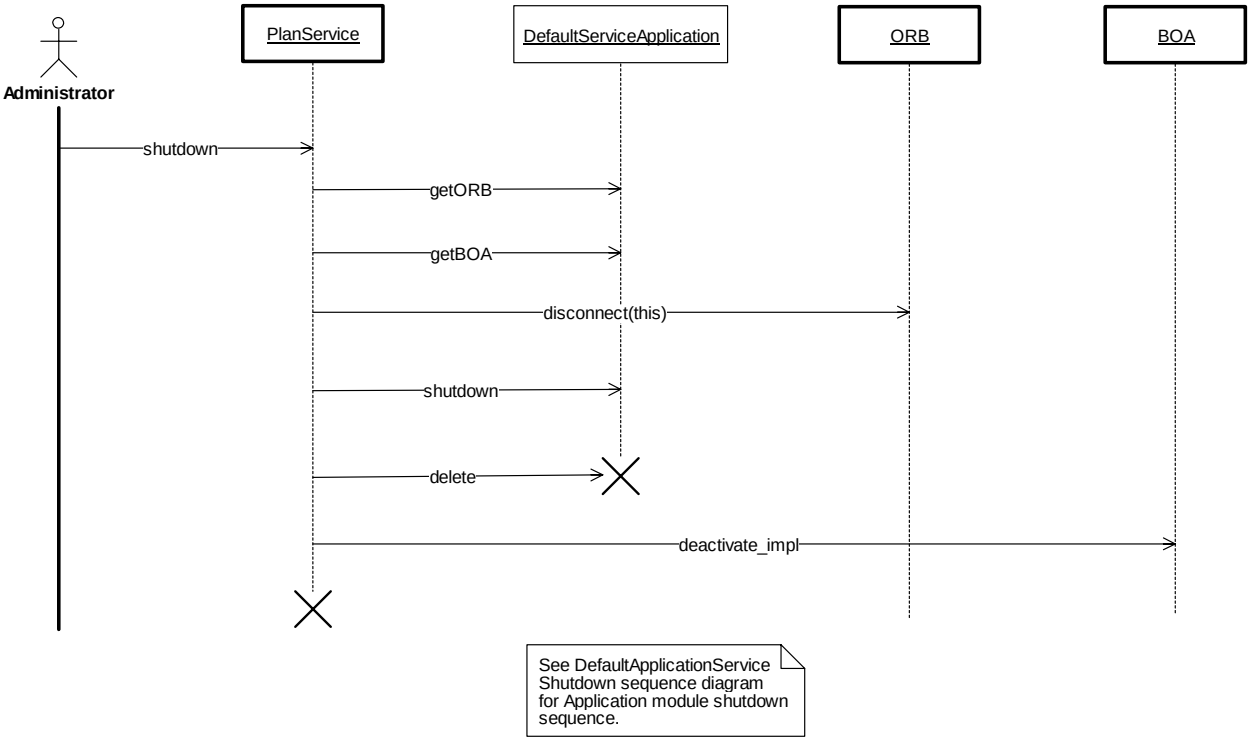

Figure 3-49. PlanService:Shutdown (Sequence Diagram)

3.5.2.2 PlanService:Startup (Sequence Diagram)

This sequence diagram shows startup of the plan service. This service creates and starts a DefaultServiceApplication object and the modules that are served by the PlanService. Refer to DefaultServiceApplication's Start sequence diagram in Utility package for details. The PlanService connects itself to the ORB and calls the method impl_is_ready on the BOA to enter the event loop and start serving the CORBA requests.

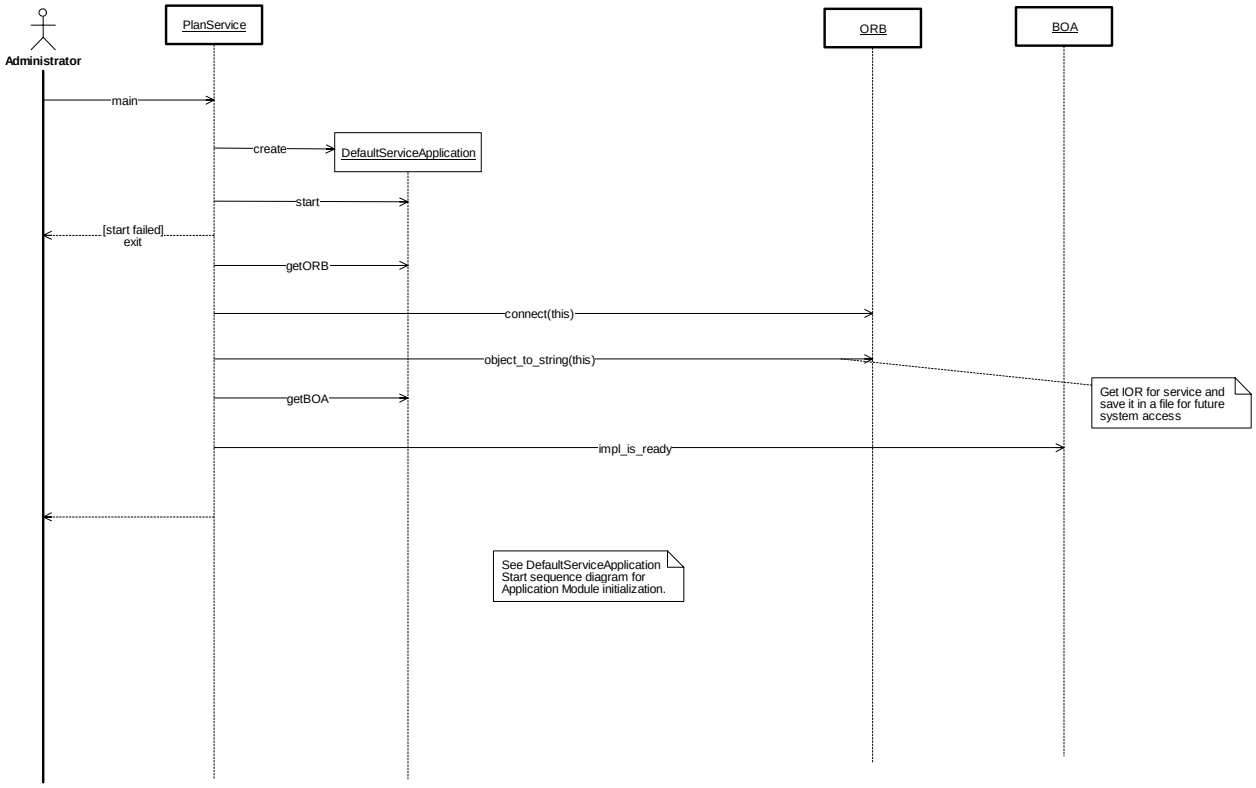

Figure 3-50. PlanService:Startup (Sequence Diagram)

3.6 PlanModule

3.6.1 PlanModuleClasses (Class Diagram)

This is an installable module that serves the PlanFactory and Plan objects to the rest of the CHART2 system.

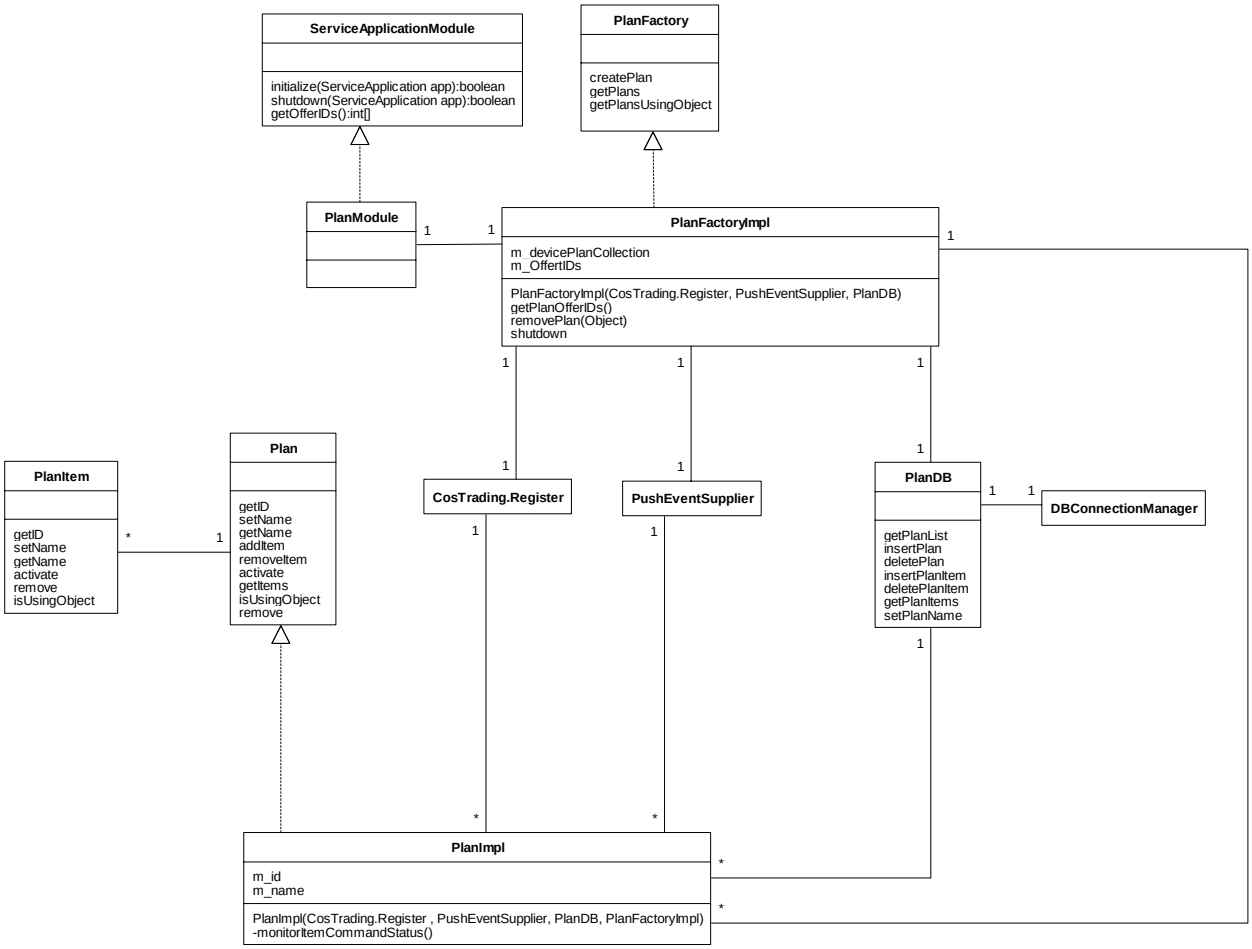

Figure 3-51. PlanModuleClasses (Class Diagram)

3.6.1.1 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

1

interface

3.6.1.2 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class

assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

1

3.6.1.3 PlanDB (Class)

This class contains the methods that perform database operations for the Plan module. It is constructed with a Database object that provides the connections to the database server. All the methods in this class get a new connection to the database before performing any operation on the database. The connection is released at completion of the operation.

3.6.1.4 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans which can be used in the system.

interface

3.6.1.5 PlanFactoryImpl (Class)

This class implements the PlanFactory interface and enables the management of the Plan objects by other processes. It creates, publishes and deletes the objects that implement the Plan interface.

3.6.1.6 Plan (Class)

This class has a collection of Plan Items which it maintains. It has functionality for changing the plan items, and also allows the plan to be activated, which has the effect of activating each plan item in the plan.

interface

3.6.1.7 PlanImpl (Class)

This class implements the Plan interface and provides the implementation for the methods defined in the interface. It also manages the database operations for the PlanItems contained in this Plan.

3.6.1.8 PlanModule (Class)

This module creates, publishes and deletes the object that implement the PlanFactory interface.

3.6.1.9 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

interface

3.6.1.10 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

1

3.6.1.11 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.6.2 Sequence Diagrams

3.6.2.1 PlanModule:ActivatePlan (Sequence Diagram)

This sequence diagram shows how a user with proper rights can activate a plan in the system. An AccessDenied exception is returned if the user does not have the functional right to activate a message. Activating the plan results in activating individual plan items. See DMSControl:ActivateDMSStoredMsgItem sequence diagrams for more details about how the PlanItems are activated. A Command status object is created to track the progress of the executed plan. This command status object is updated periodically and is destroyed when the plan execution is completed. Similar Command Status objects are created for each of the plan items. The user is informed of the progress of the plan execution through these Command Status objects.

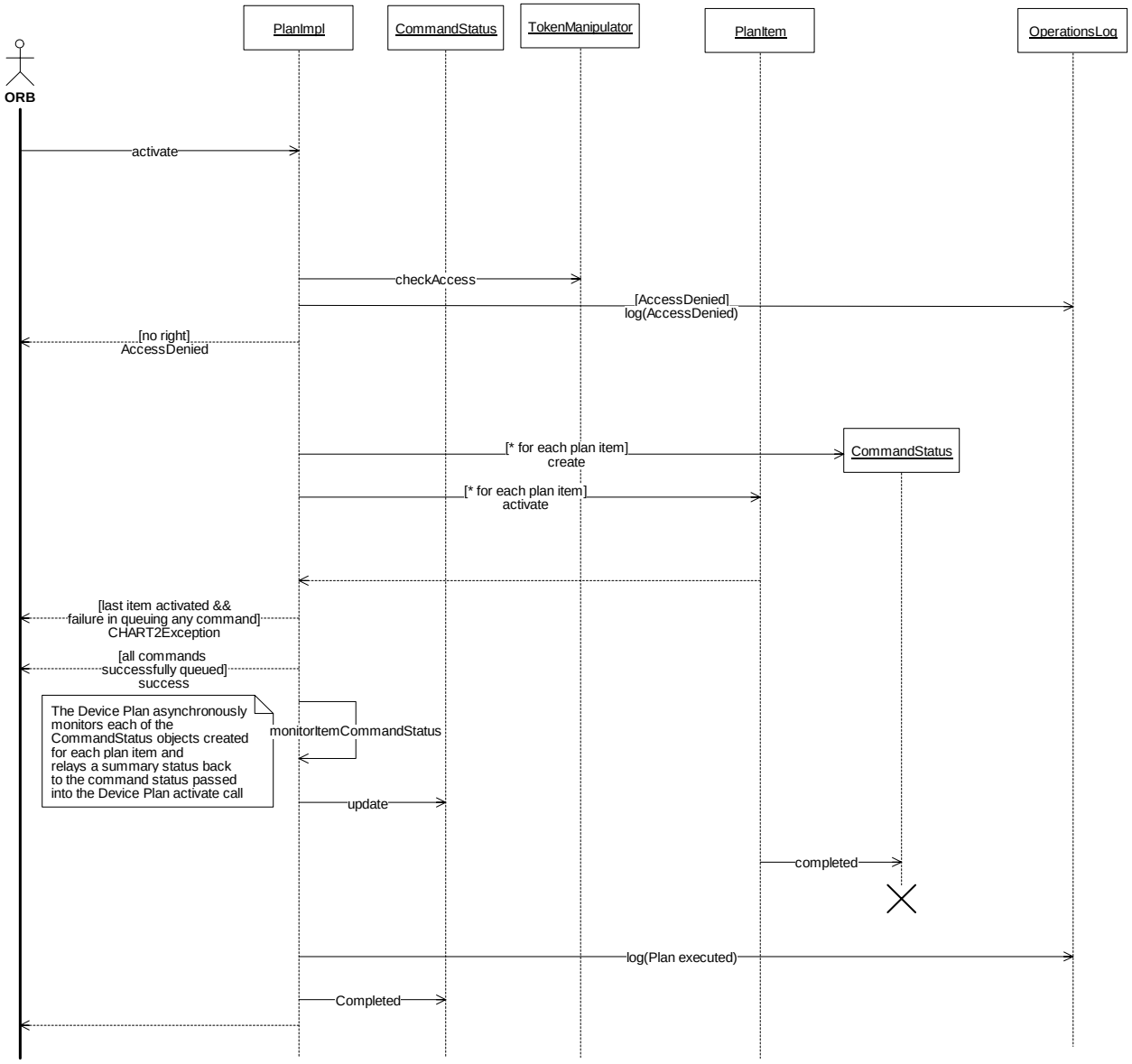

Figure 3-52. PlanModule:ActivatePlan (Sequence Diagram)

3.6.2.2 PlanModule:AddItem (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can add an item to an existing plan in the system. An AccessDenied exception is returned if the user does not have the right to add an item to the plan. Otherwise, a PlanItem object is created and added to the database. A PlanItemAdded event is pushed through the event channel to notify other processes that a plan item has been added to this plan. User actions are logged to the operations log.

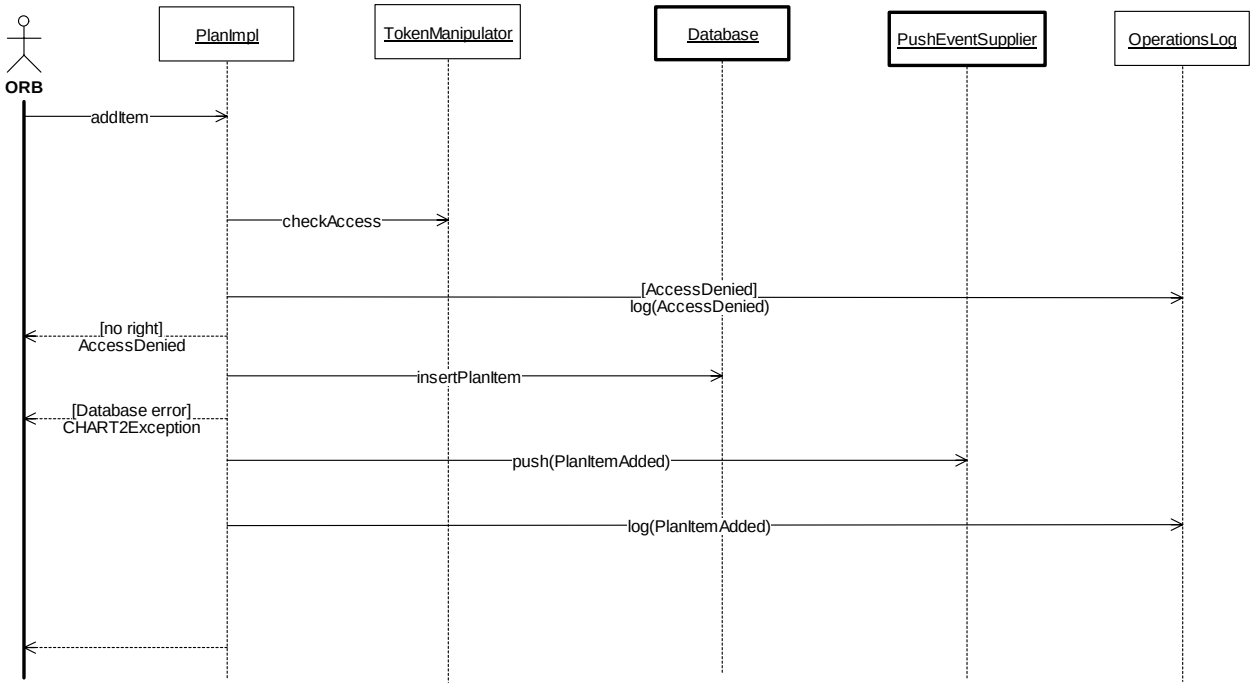

Figure 3-53. PlanModule:AddItem (Sequence Diagram)

3.6.2.3 PlanModule:AddPlan (Sequence Diagram)

This diagram shows how a user with proper functional rights can add a plan to the system. An AccessDenied exception is returned if the user does not have the functional right to add a plan. Otherwise, the plan object is created and added to the database. The plan object is published in CORBA Trader service and a PlanAdded event is pushed through the event channel to notify the other processes that a new plan has been added.

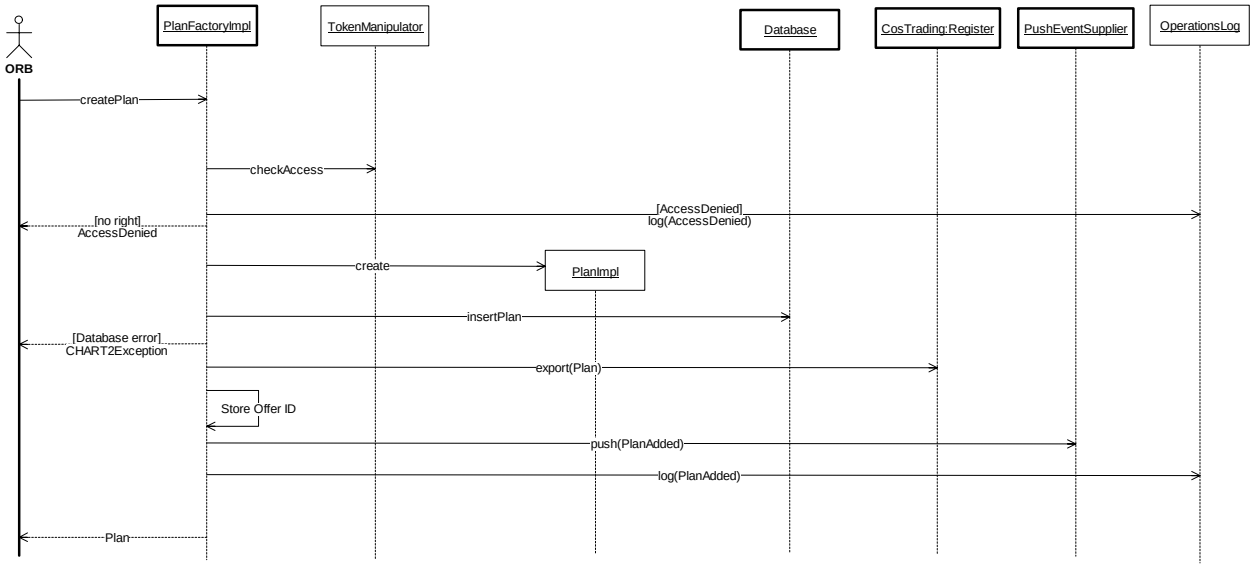

Figure 3-54. PlanModule:AddPlan (Sequence Diagram)

3.6.2.4 PlanModule:GetPlansUsingObject (Sequence Diagram)

This sequence diagrams shows how to get a list of Plans that are using a particular object. The ID of the object is passed to the Plan object to check if its PlanItems are using this object. If a PlanItem is using the object, the Plan is added to a list. The list of Plans is returned after all the Plans are checked.

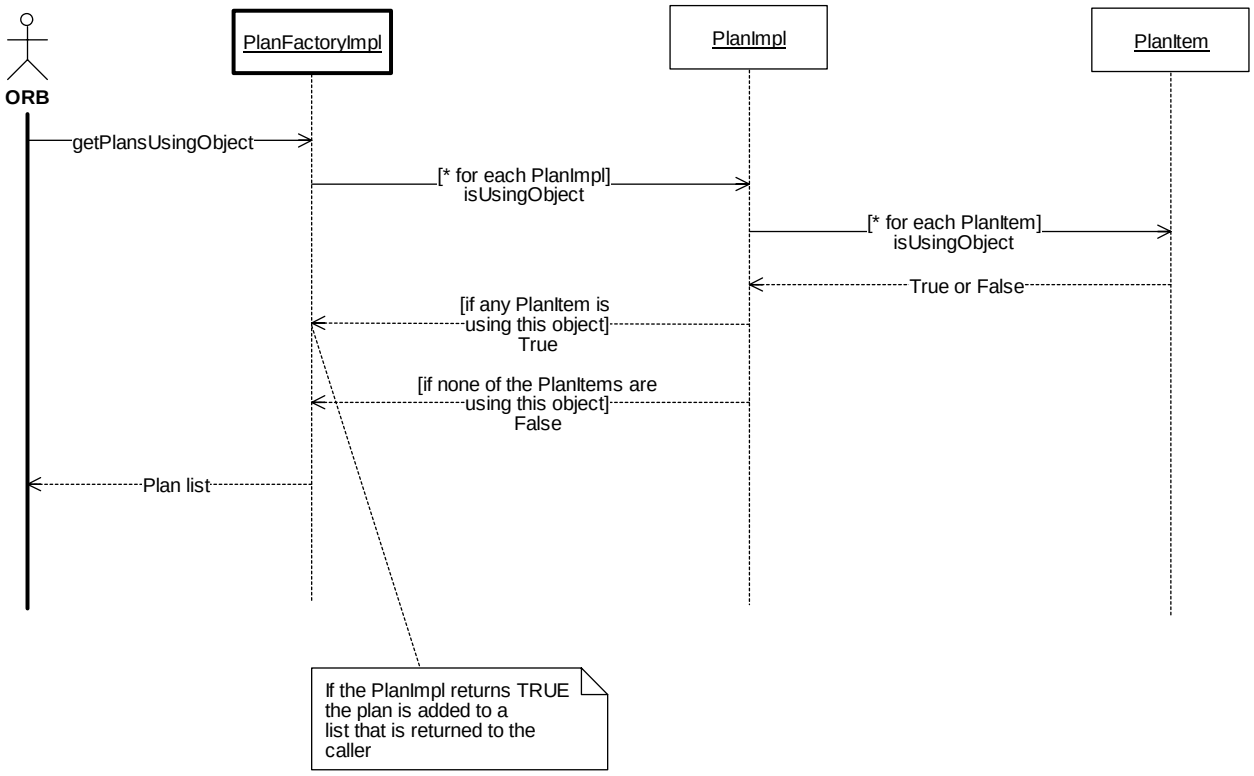

Figure 3-55. PlanModule:GetPlansUsingObject (Sequence Diagram)

3.6.2.5 PlanModule:Initialize (Sequence Diagram)

This sequence diagram shows the startup for the Plan Module. An ApplicationService will initialize this module. The connections to basic services such as ORB, Trader, Event channel and database are obtained from the ServiceApplication. This module creates a Plan Module specific database object. It also creates the PlanFactory object, which creates the Plan objects from the plan list obtained from the database. The Plan objects are published in the trader. An event channel is created to push the events to clients and it is published in the trader register. The Offer IDs of all the objects that were published in the trader are saved to a file so that they may be withdrawn.

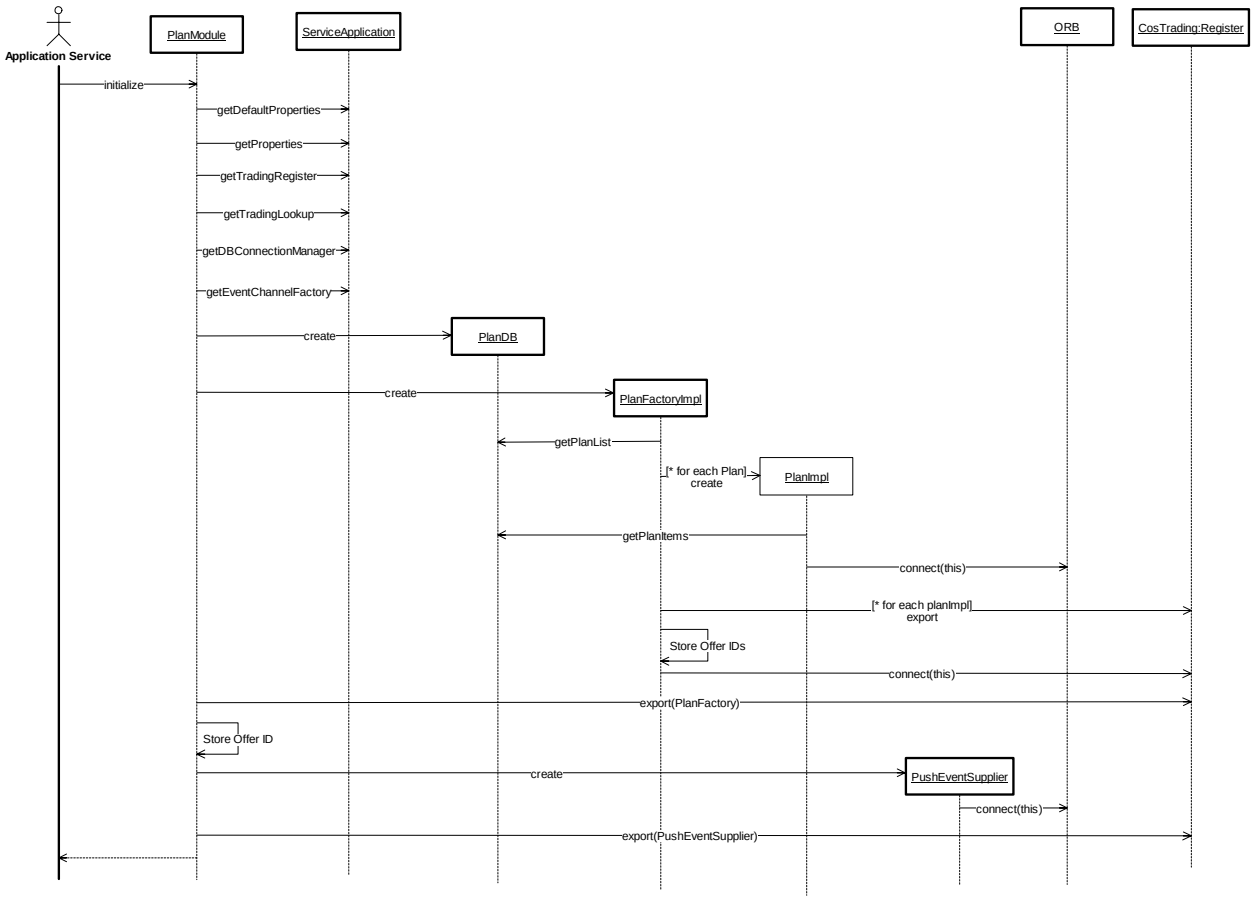

Figure 3-56. PlanModule:Initialize (Sequence Diagram)

3.6.2.6 PlanModule:RemoveItem (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can remove a plan item from a plan in the system. An AccessDenied exception is returned if the user does not have the right to remove an item from the plan. Otherwise, the plan item is deleted from the database and the object is destroyed. An event is pushed through the event channel to notify other processes that the plan item has been removed from the plan. User actions are logged to the operations log.

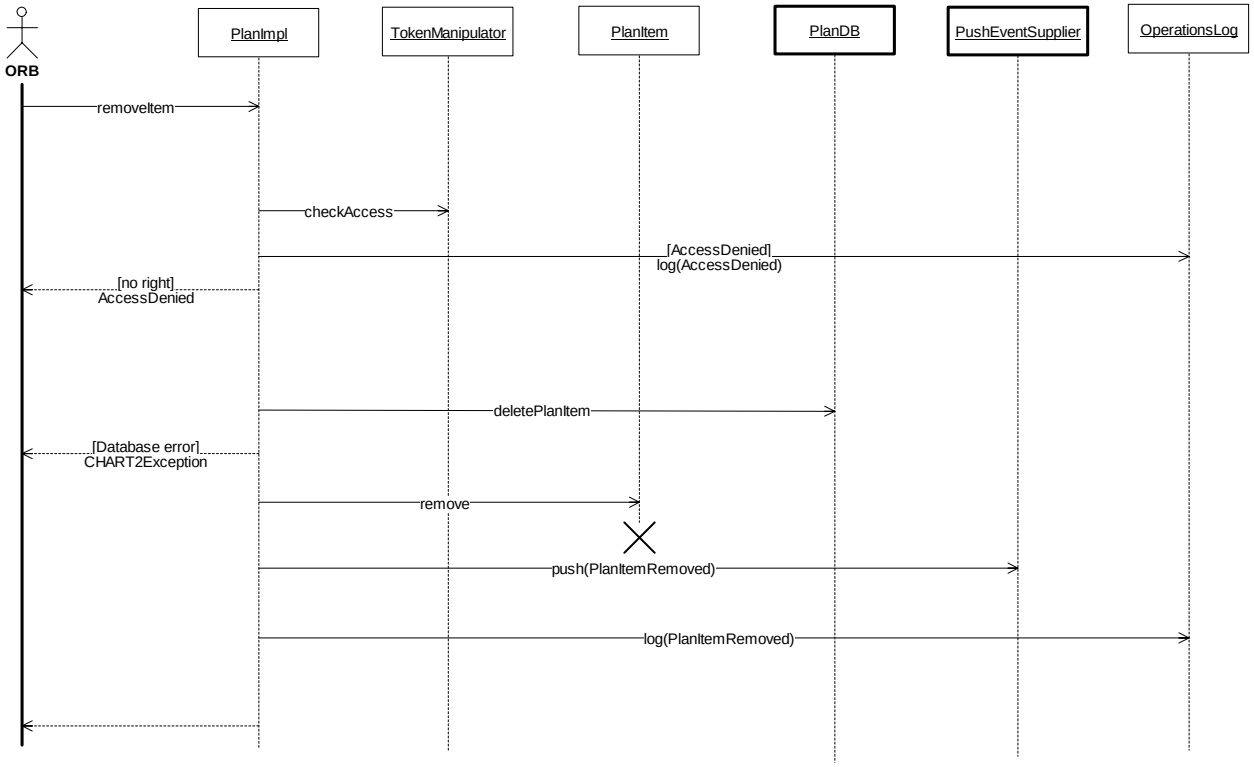

Figure 3-57. PlanModule:RemoveItem (Sequence Diagram)

3.6.2.7 PlanModule:RemovePlan (Sequence Diagram)

This sequence diagram shows how a user with proper rights can delete a Plan from the system. An AccessDenied exception is returned if the user does not have the functional right to delete a Plan. Otherwise, the Plan is deleted from the database and the object is destroyed. The Plan is withdrawn from the trader and a PlanRemoved event is pushed through the event channel to notify the clients that the plan has been deleted. Note that the deletion of a plan results in the deletion of all the plan items that are used in the plan from the system and the database. The user actions are logged to the operations log.

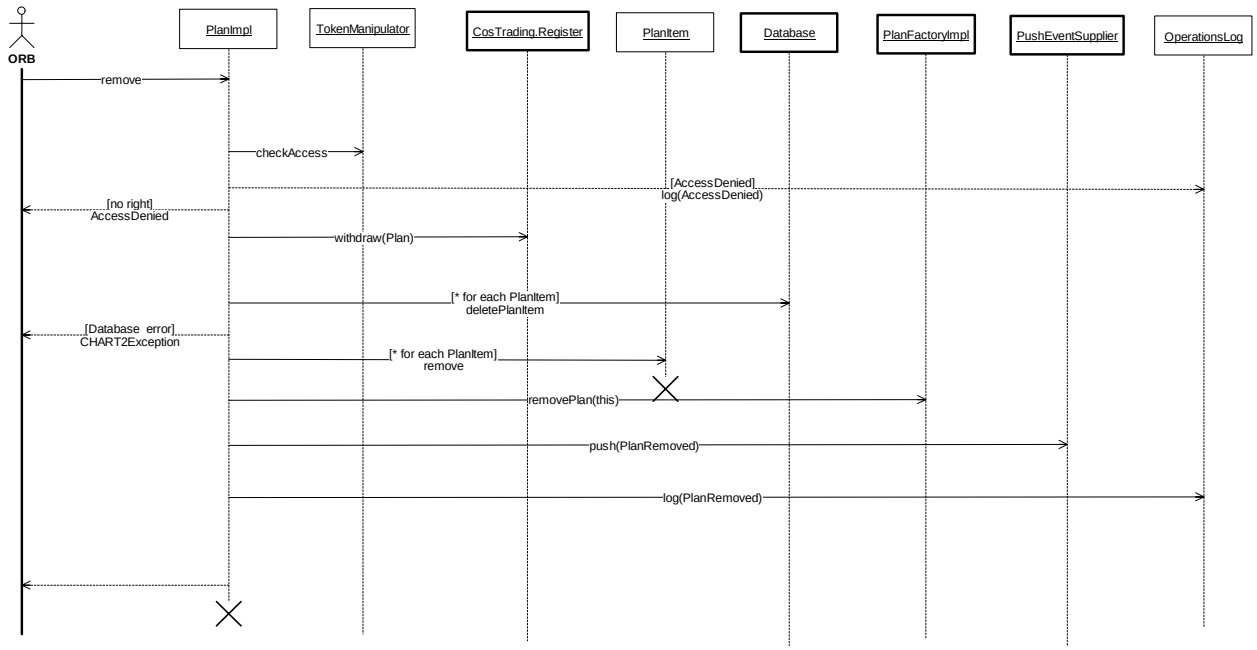

Figure 3-58. PlanModule:RemovePlan (Sequence Diagram)

3.6.2.8 PlanModule:RemovePlanFromFactory (Sequence Diagram)

This sequence diagram shows how a Plan object is removed from the Plan Factory when a Plan is deleted from the system.

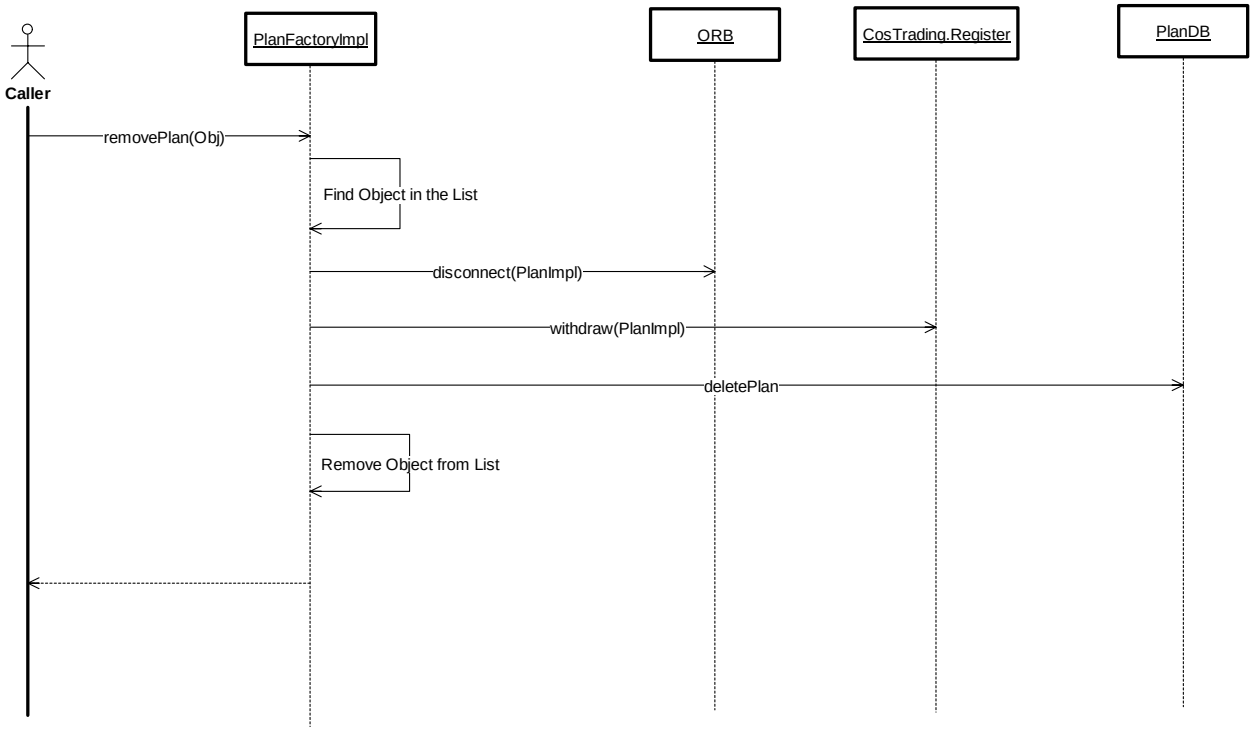

Figure 3-59. PlanModule:RemovePlanFromFactory (Sequence Diagram)

3.6.2.9 PlanModule:SetPlanName (Sequence Diagram)

This sequence diagram shows how a user with proper functional rights can set the name of a Plan. An access denied exception is returned if the user does not have the right to change the name. Otherwise, the name is changed and the database is updated. An event id pushed via the CORBA event service to notify others of the new Plan name. The user actions are logged to the operations log.

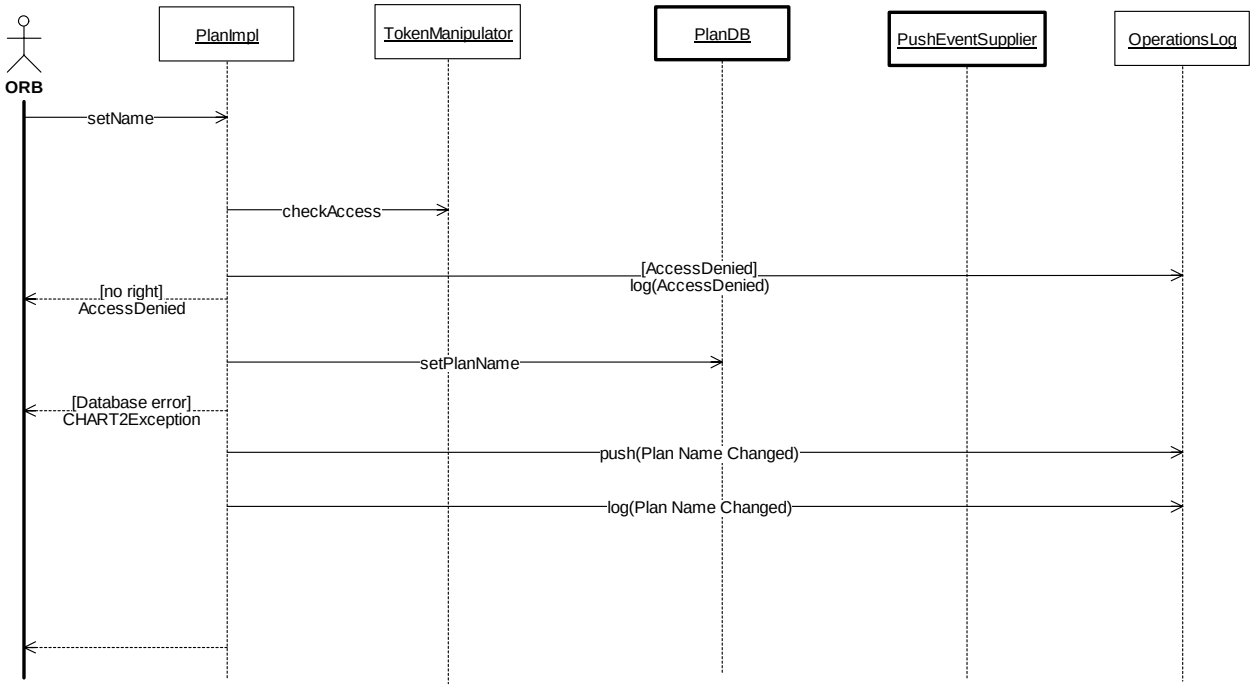

Figure 3-60. PlanModule:SetPlanName (Sequence Diagram)

3.6.2.10 PlanModule:Shutdown (Sequence Diagram)

This diagram shows the shutdown sequence of the Plan module. All the Plan objects that were published in the trader by the PlanFactory and the PlanFactory itself are withdrawn and destroyed. The event channel is also withdrawn from the trader and destroyed.

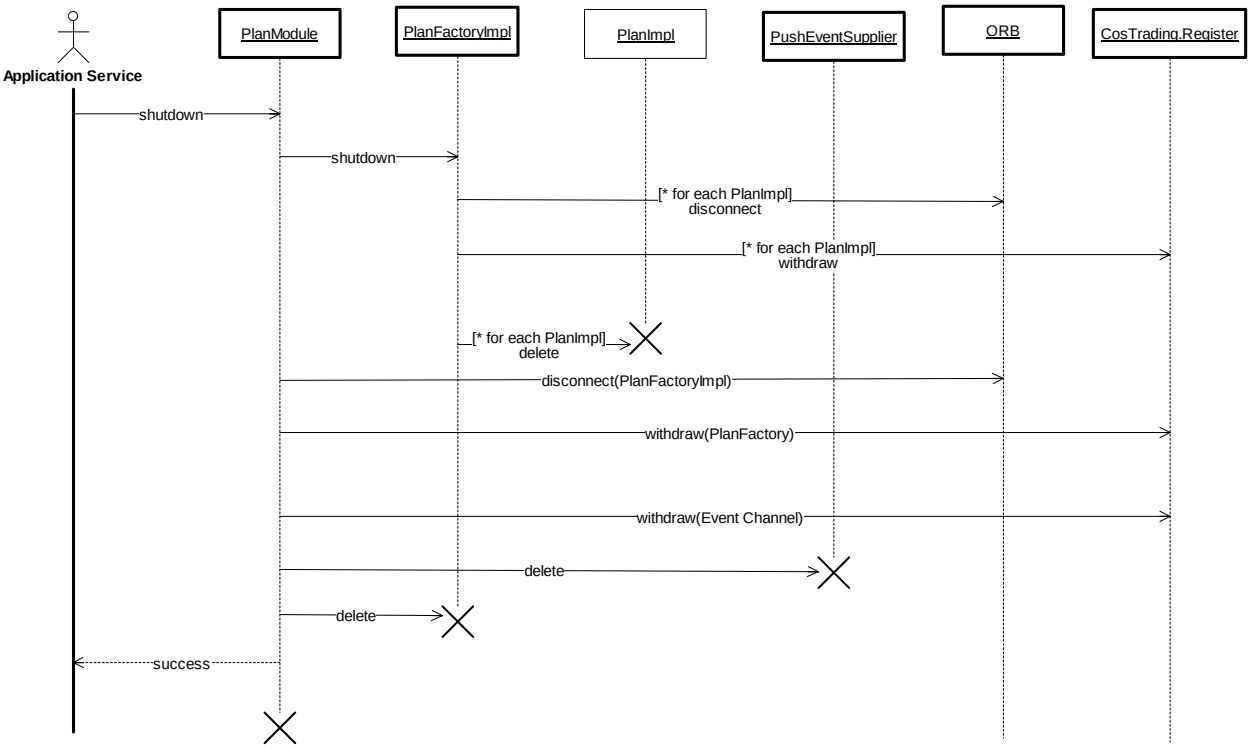

Figure 3-61. PlanModule:Shutdown (Sequence Diagram)

3.7 UserManagementService

3.7.1 UserManagementServiceClassDiagram (Class Diagram)

This diagram models the class relationships that exist within the User management service application. Note that the UserManagementService does not contain any classes that are specific to the UserManagementModule or the UserManagementResourcesModule. It knows only that it will utilize a DefaultServiceApplication that will aid it in determining which modules need to be installed, and setting up the CORBA ORB, CORBA services, and system database connection. The details pertinent to the installed modules are included in separate models.

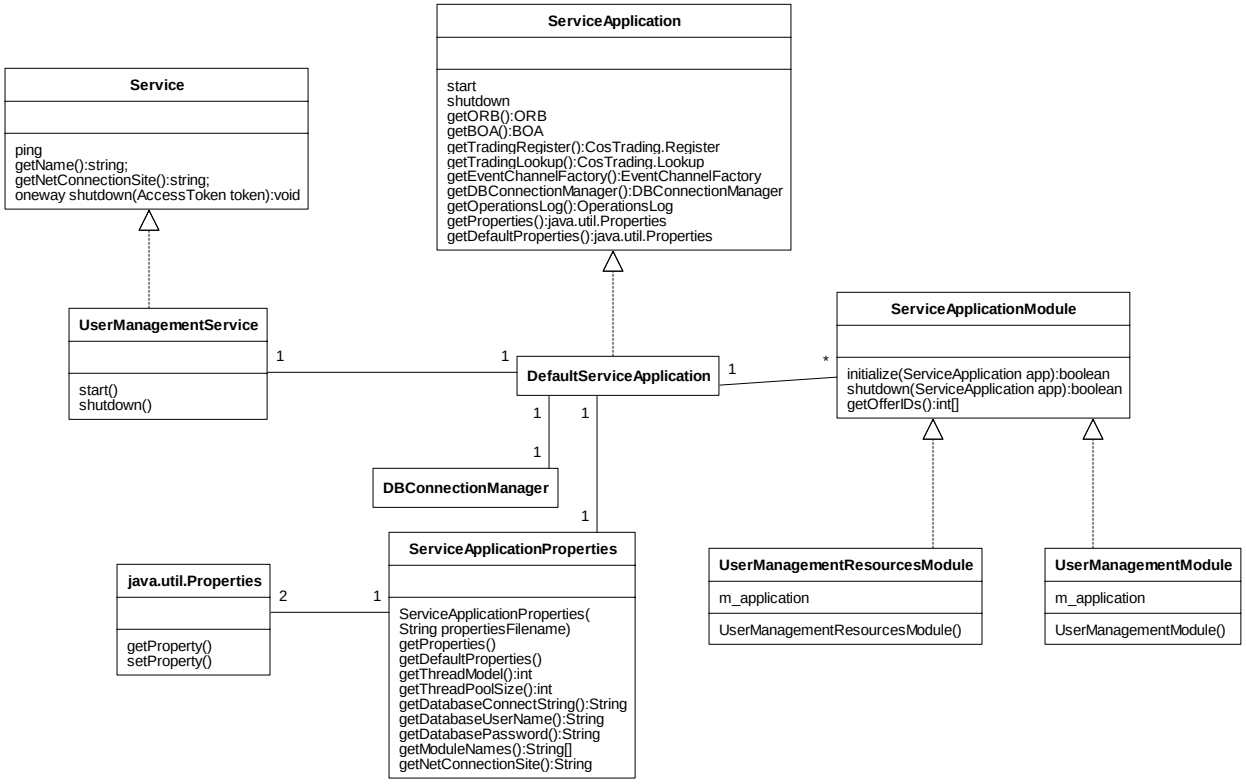

Figure 3-62. UserManagementServiceClassDiagram (Class Diagram)

3.7.1.1 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.7.1.2 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the

DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.7.1.3 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults;" this second property list is searched if the property key is not found in the original property list.

3.7.1.4 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

interface

3.7.1.5 ServiceApplicationProperties (Class)

This class provides methods that allow the DefaultServiceApplication to access the necessary properties from the java properties configuration file. It also provides a default properties file which can be retrieved by anyone holding a ServiceApplication interface reference. This gives each installed service module the opportunity to load default values before retrieving property values from the properties file.

3.7.1.6 UserManagementModule (Class)

This module creates, publishes and deletes the object that implements the UserManager interface for user configuration and manipulation.

3.7.1.7 UserManagementResourcesModule (Class)

This module creates, publishes and destroys all objects related to resource management that are used by the User Management service application.

3.7.1.8 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.7.1.9 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.7.1.10 UserManagementService (Class)

This is the main class for the User Management service application. The main method in this class will be the main entry point for this application. The application simply provides basic CORBA services to the rest of the application and manages the initialization and shutdown of the installed modules.

3.7.2 Sequence Diagrams

3.7.2.1 UserManagementService:Shutdown (Sequence Diagram)

The user management service is responsible for shutting down the application. It does this by shutting down the DefaultServiceApplication which will not return until all of the installable modules have been shutdown. At this point, the service will call the deactivate_impl method on the basic object adapter (BOA). The deactivate_impl call frees the main thread that is blocking on the impl_is_ready call within the main function of the application. This will cause the application to exit by reaching the end of the main method.

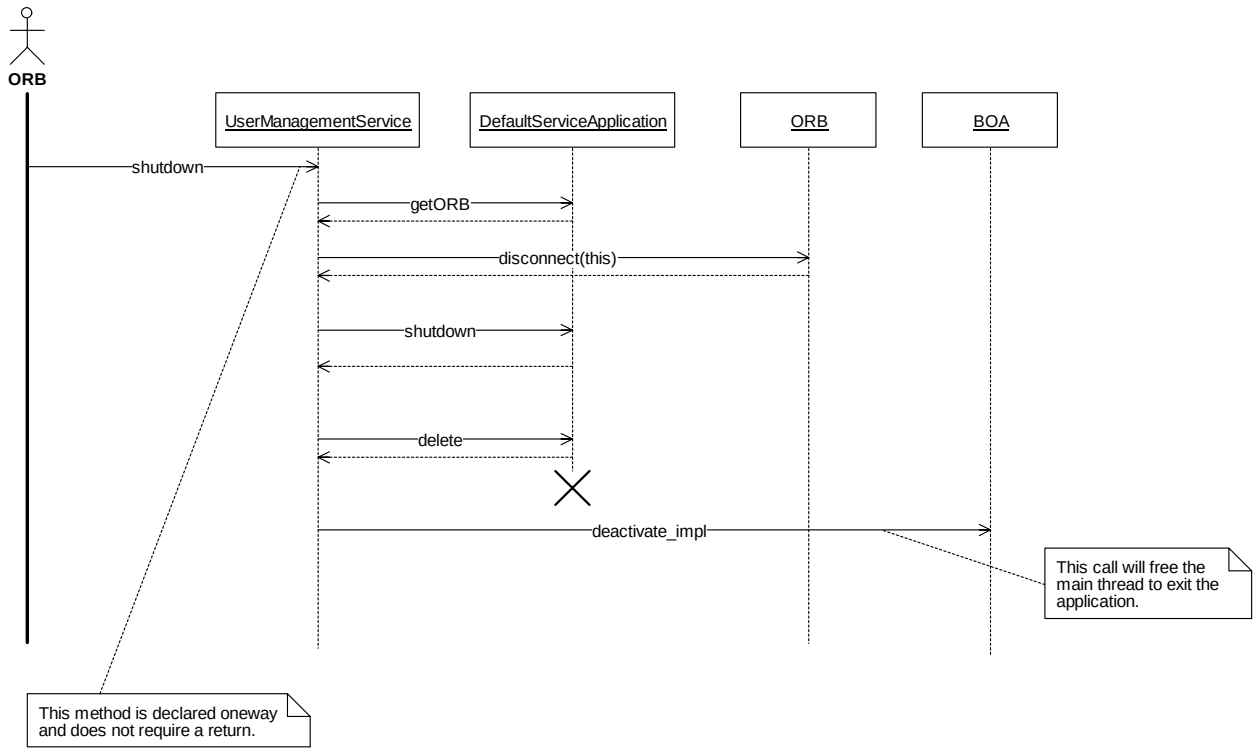

Figure 3-63. UserManagementService:Shutdown (Sequence Diagram)

3.7.2.2 UserManagementService:Startup (Sequence Diagram)

The user management service class implements the Service IDL interface. As such, it must be connected to the ORB. Additionally, it is the responsibility of the service object to create the DefaultServiceApplication object that will install all of the installable service modules. Finally, a call to BOA.impl_is_ready will be made. This call will block until the deactivate_impl is called at which point the application will exit.

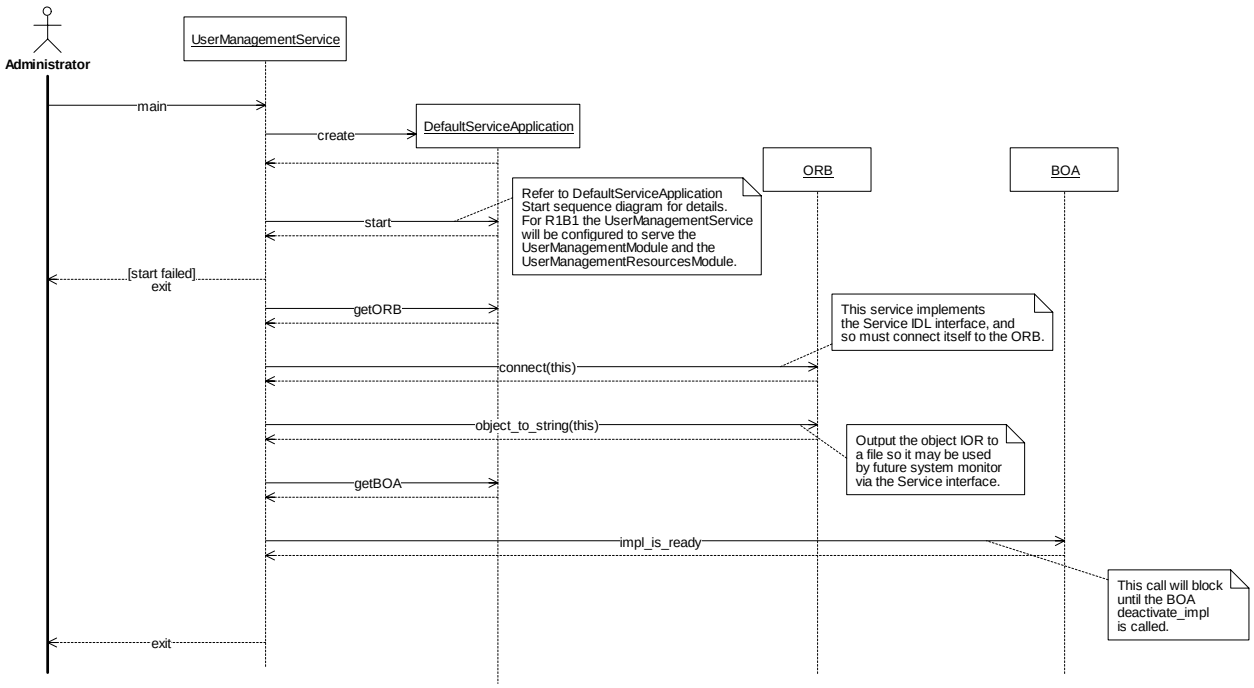

Figure 3-64. UserManagementService:Startup (Sequence Diagram)

3.8 UserManagementModule

3.8.1 UserManagementModuleClasses (Class Diagram)

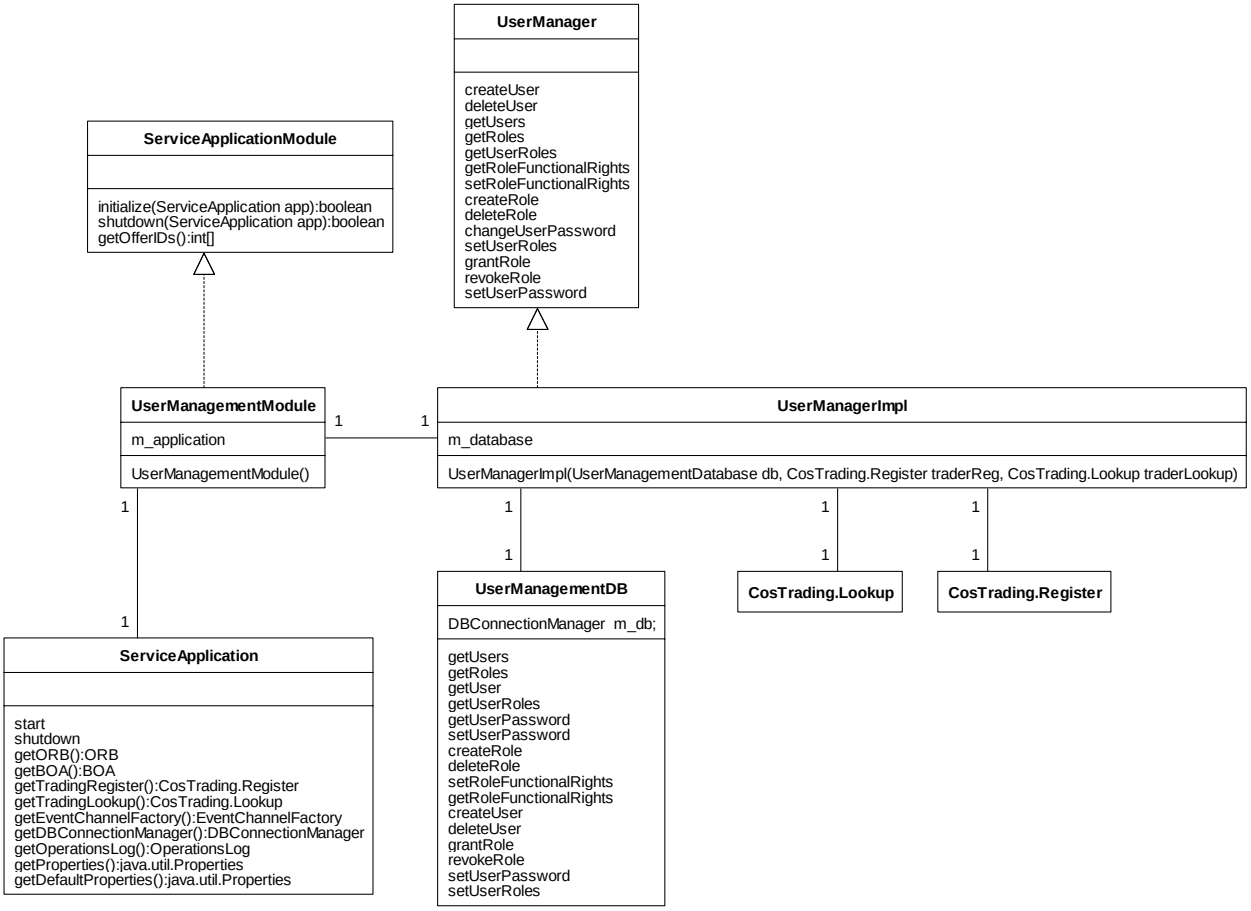

Figure 3-65. UserManagementModuleClasses (Class Diagram)

3.8.1.1 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

1

interface

3.8.1.2 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.8.1.3 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.8.1.4 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects which have previously been published.

1

interface

3.8.1.5 UserManagementDB (Class)

The UserManagementDB Class provides methods used to access and modify User Managment data in the database. This class uses a Database object to retrieve a connection to the database for its exclusive use during a method call.

3.8.1.6 UserManagementModule (Class)

This module creates, publishes and deletes the object that implements the UserManager interface for user configuration and manipulation.

3.8.1.7 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

1

interface

3.8.1.8 UserManagerImpl (Class)

This class is the specific implementation of a UserManager interface which will be served by the User Management Service. As such, it provides implementations of each of the methods in the UserManger interface.

3.8.2 Sequence Diagrams

3.8.2.1 UserManagementModule:AddUser (Sequence Diagram)

A user with the proper functional rights may add a new user to the system. The user will be added to the user database provided the password and username specified for the new user are valid.

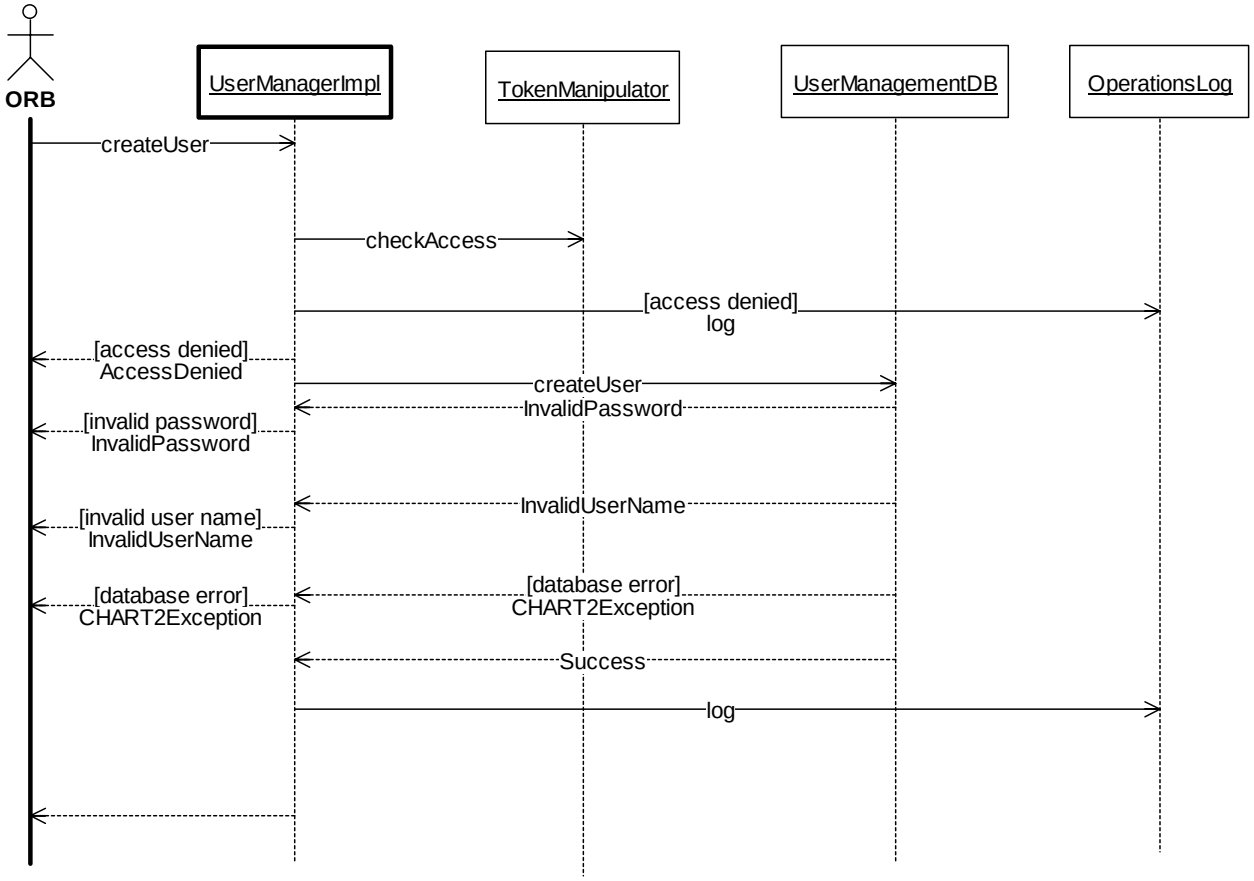

Figure 3-66. UserManagementModule:AddUser (Sequence Diagram)

3.8.2.2 UserManagementModule:ChangeUserPassword (Sequence Diagram)

A user may change his/her own password. The system will verify that the invoking user is actually the user whose password is being changed and will require the user to pass his/her current password which must match the password in the user database.

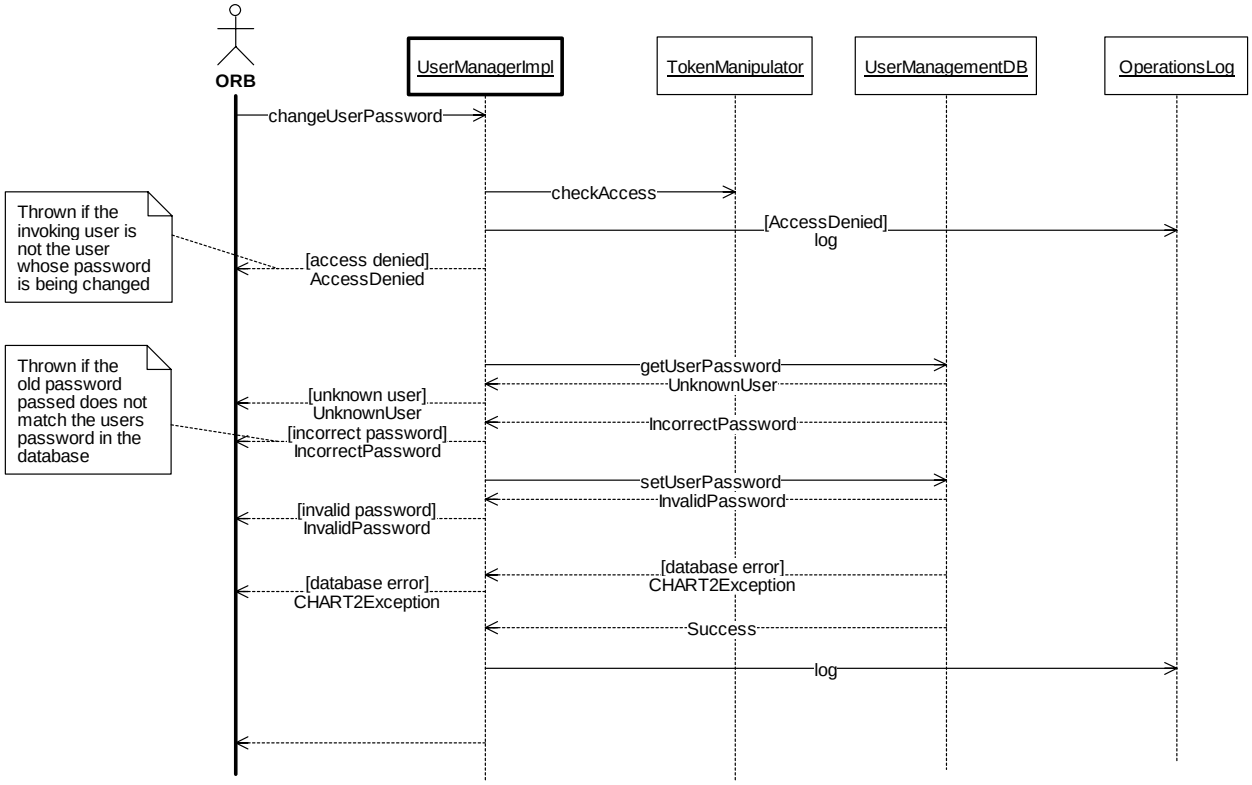

Figure 3-67. UserManagementModule:ChangeUserPassword (Sequence Diagram)

3.8.2.3 UserManagementModule:CreateRole (Sequence Diagram)

A user with the proper functional rights may create a new role in the user database. The system will verify that the role is not already defined before creating it.

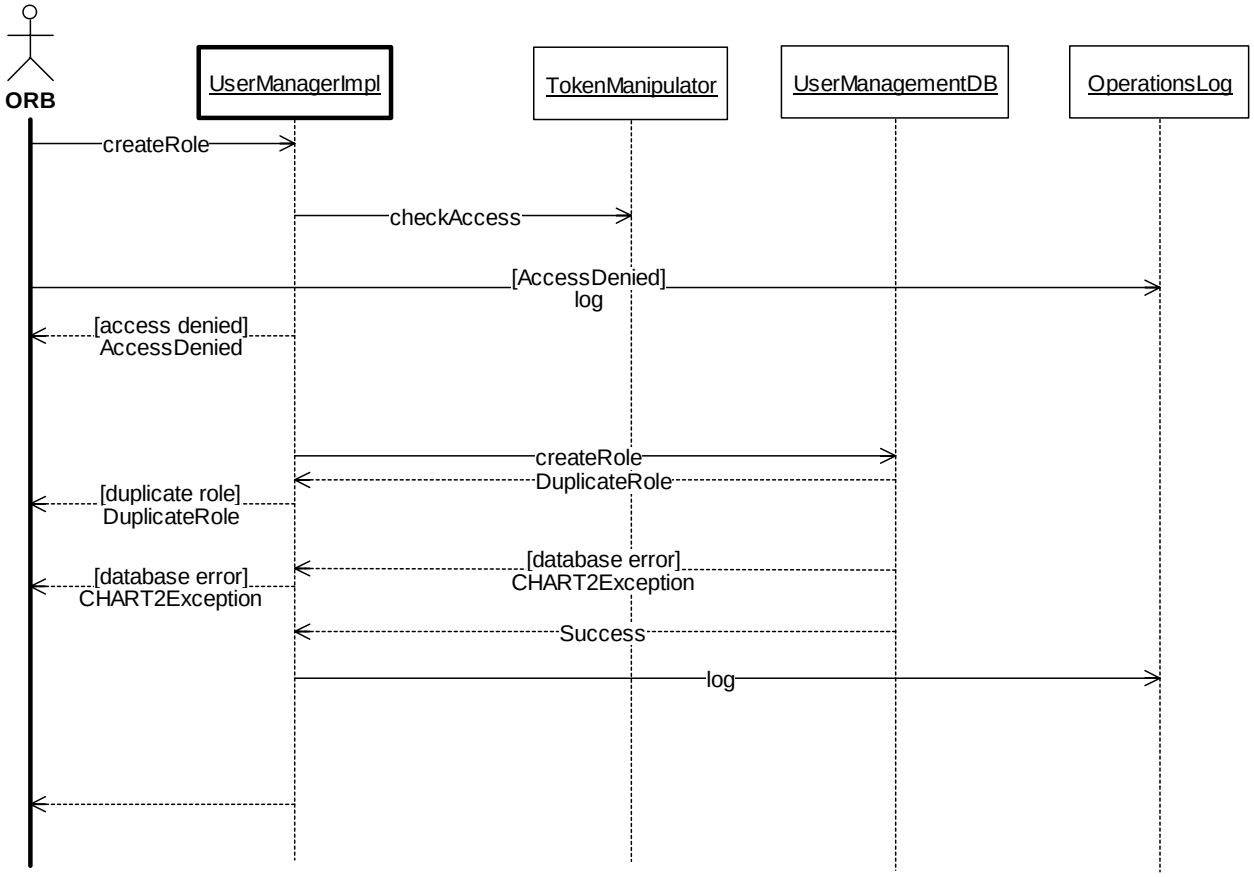

Figure 3-68. UserManagementModule:CreateRole (Sequence Diagram)

3.8.2.4 UserManagementModule:DeleteRole (Sequence Diagram)

A user with the proper functional rights may delete a role from the user database. The system will verify that the role is not currently assigned to any users before deleting it.

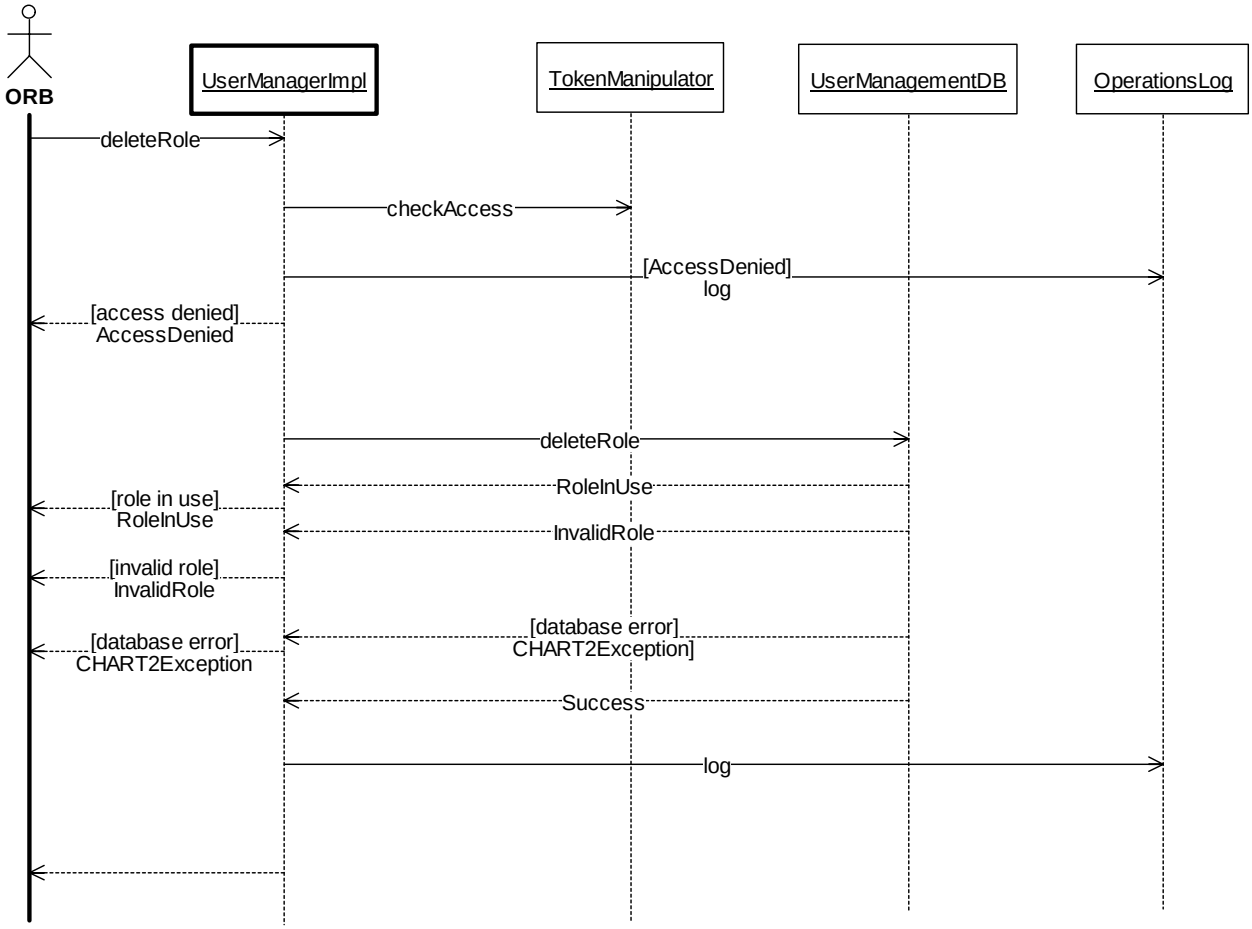

Figure 3-69. UserManagementModule:DeleteRole (Sequence Diagram)

3.8.2.5 UserManagementModule:DeleteUser (Sequence Diagram)

A user with the proper functional rights may delete a user from the user database. The system will check if the user who is being deleted is currently logged in. If the user is logged in, the administrator will be notified of this fact and will not be able to delete the user. Note that the administrator may use the system to force the user to logout and then delete the user. The check to see if the user is currently logged in is a warning to the administrator and, due to its use of the trader, cannot be guaranteed to successfully check all logins. If the user is deleted from the database while logged in, however, it will not affect his/her current session. He/she will simply not be able to use the system subsequent to logging out.

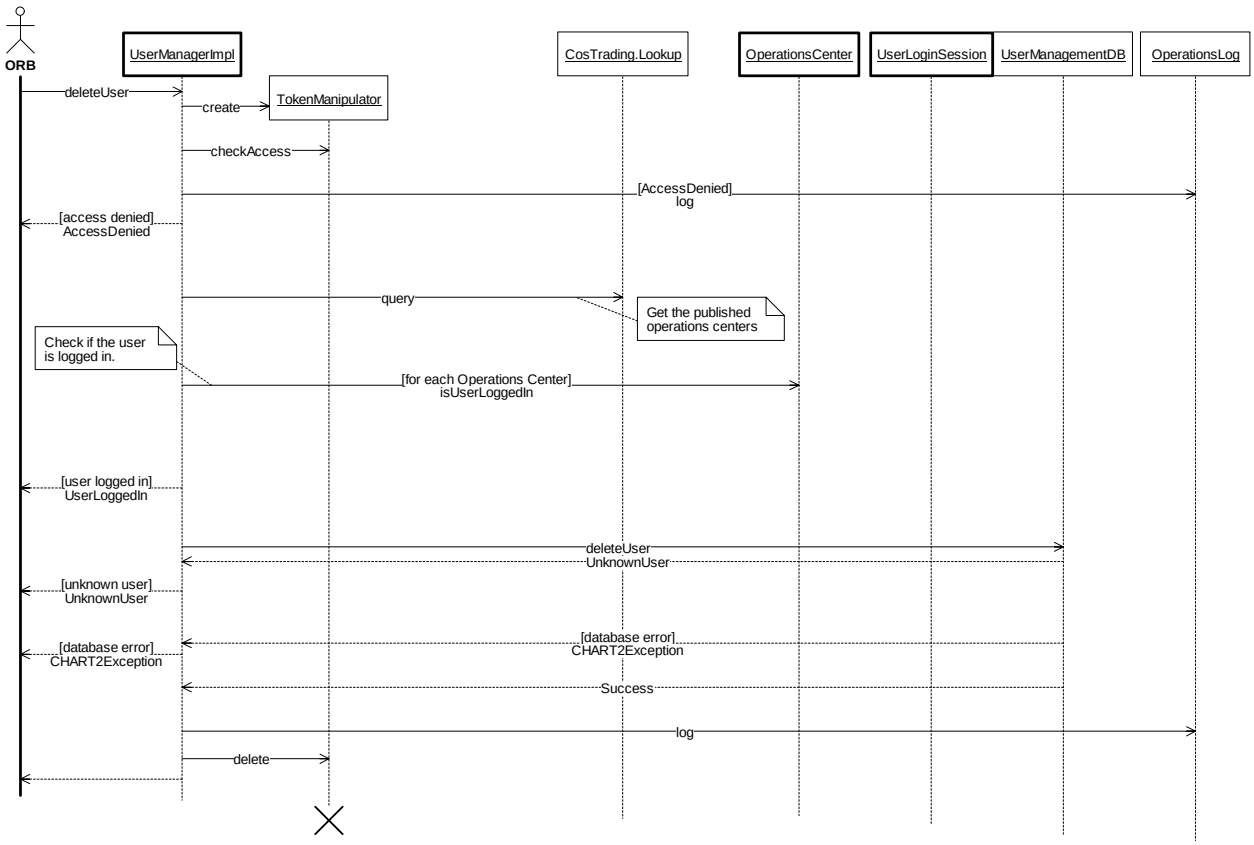

Figure 3-70. UserManagementModule:DeleteUser (Sequence Diagram)

3.8.2.6 UserManagementModule:GrantRole (Sequence Diagram)

A user with the proper functional rights may grant a role to a user. The user will not get his/her new functional rights until he/she logs off and logs back on.

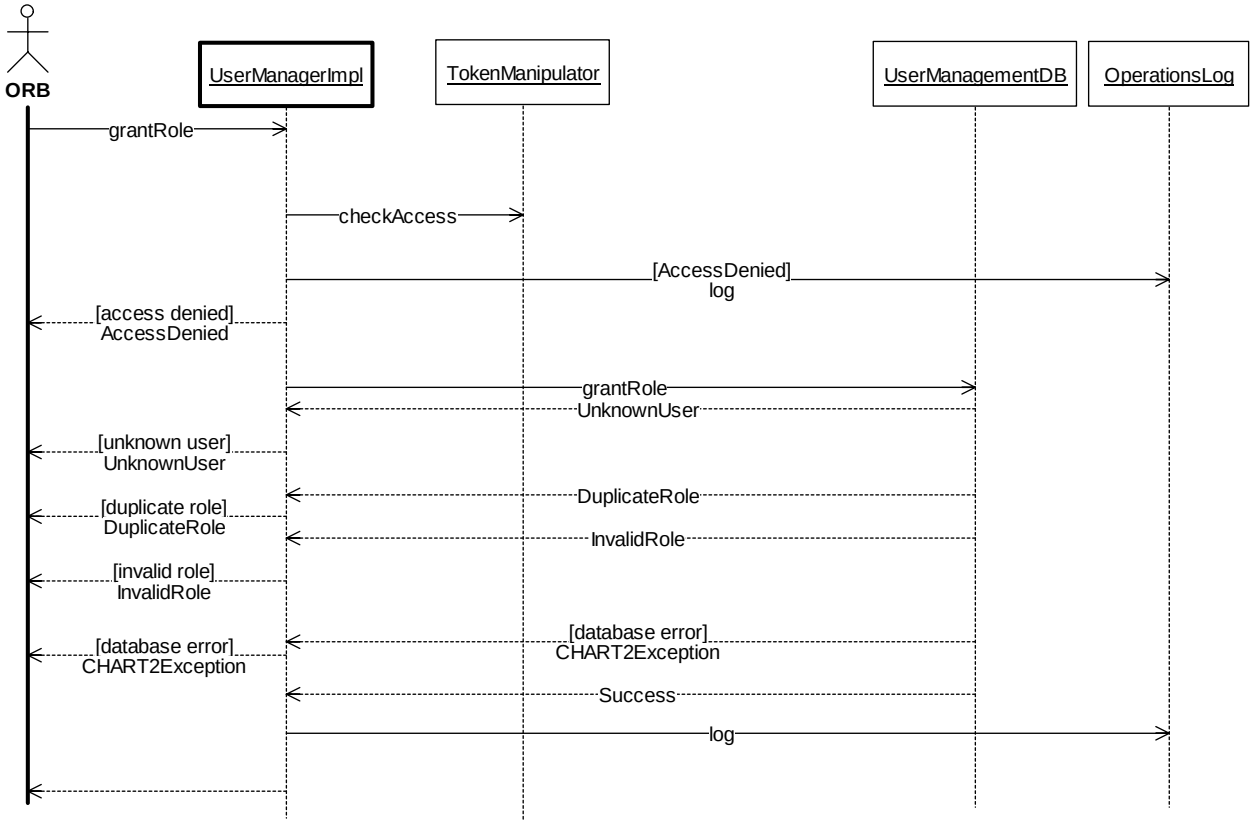

Figure 3-71. UserManagementModule:GrantRole (Sequence Diagram)

3.8.2.7 UserManagementModule:Initialize (Sequence Diagram)

Upon initialization the user manager module will create the objects that it is responsible for serving, connect them to the ORB, and export them to the CORBA trading service. After initialization this module is available for use by clients.

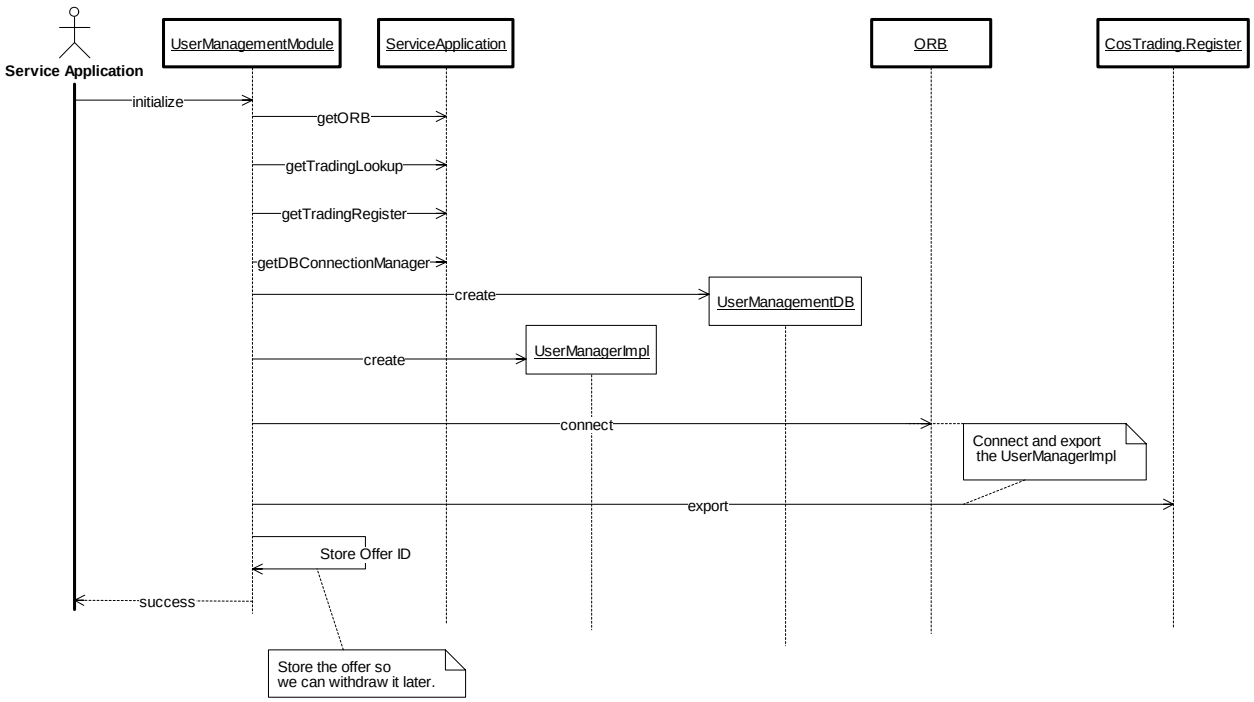

Figure 3-72. UserManagementModule:Initialize (Sequence Diagram)

3.8.2.8 UserManagementModule:ModifyRole (Sequence Diagram)

A user with the proper functional rights may change the functional rights that belong to a role. This will have the effect of changing the actions that users who have been granted that role may perform. However, these changes will not be recognized until the user logs out and logs back in.

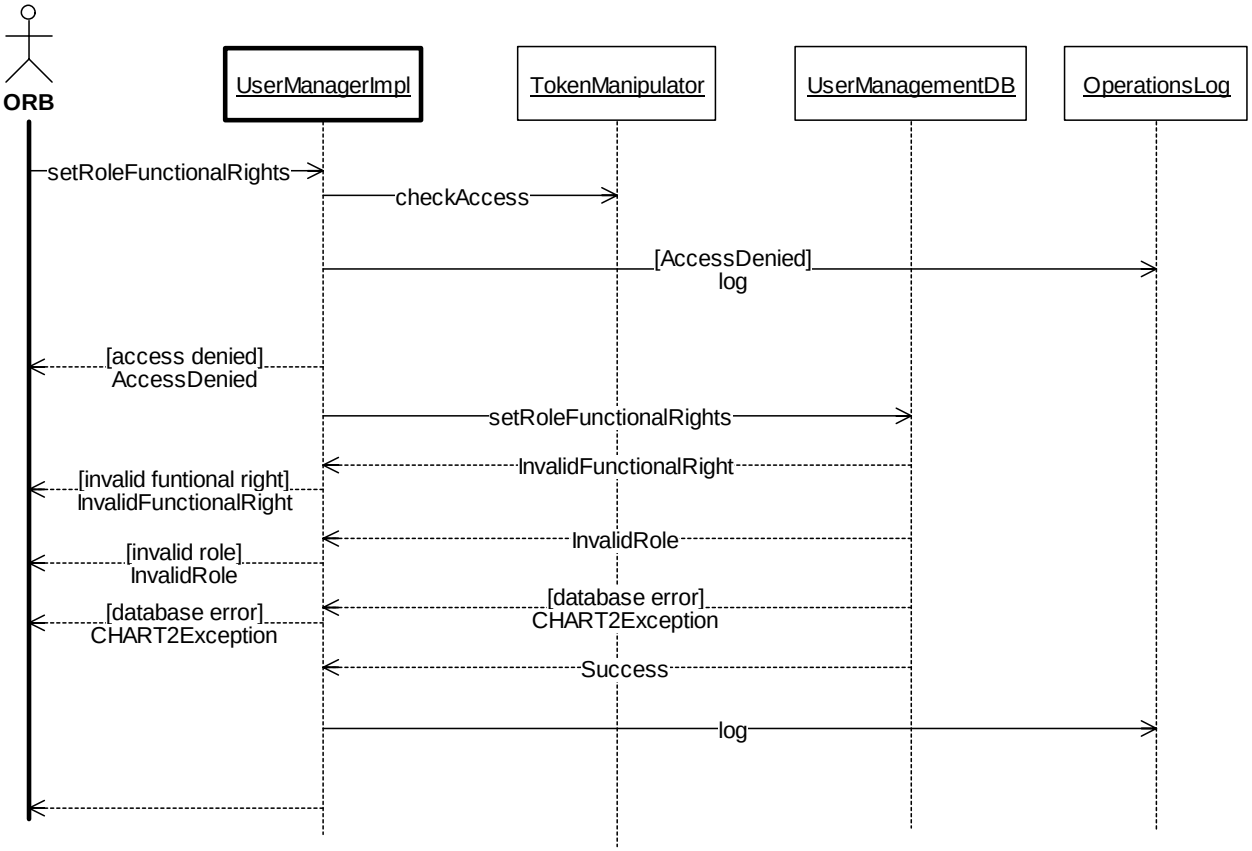

Figure 3-73. UserManagementModule:ModifyRole (Sequence Diagram)

3.8.2.9 UserManagementModule:RevokeRole (Sequence Diagram)

A user with the proper functional rights may revoke a role that has previously been granted to a user. This action will result in the user having a reduced set of functional rights, and thus reduce the number of system activities the user may perform. The user will get his/her new list of functional rights the next time he/she logs in.

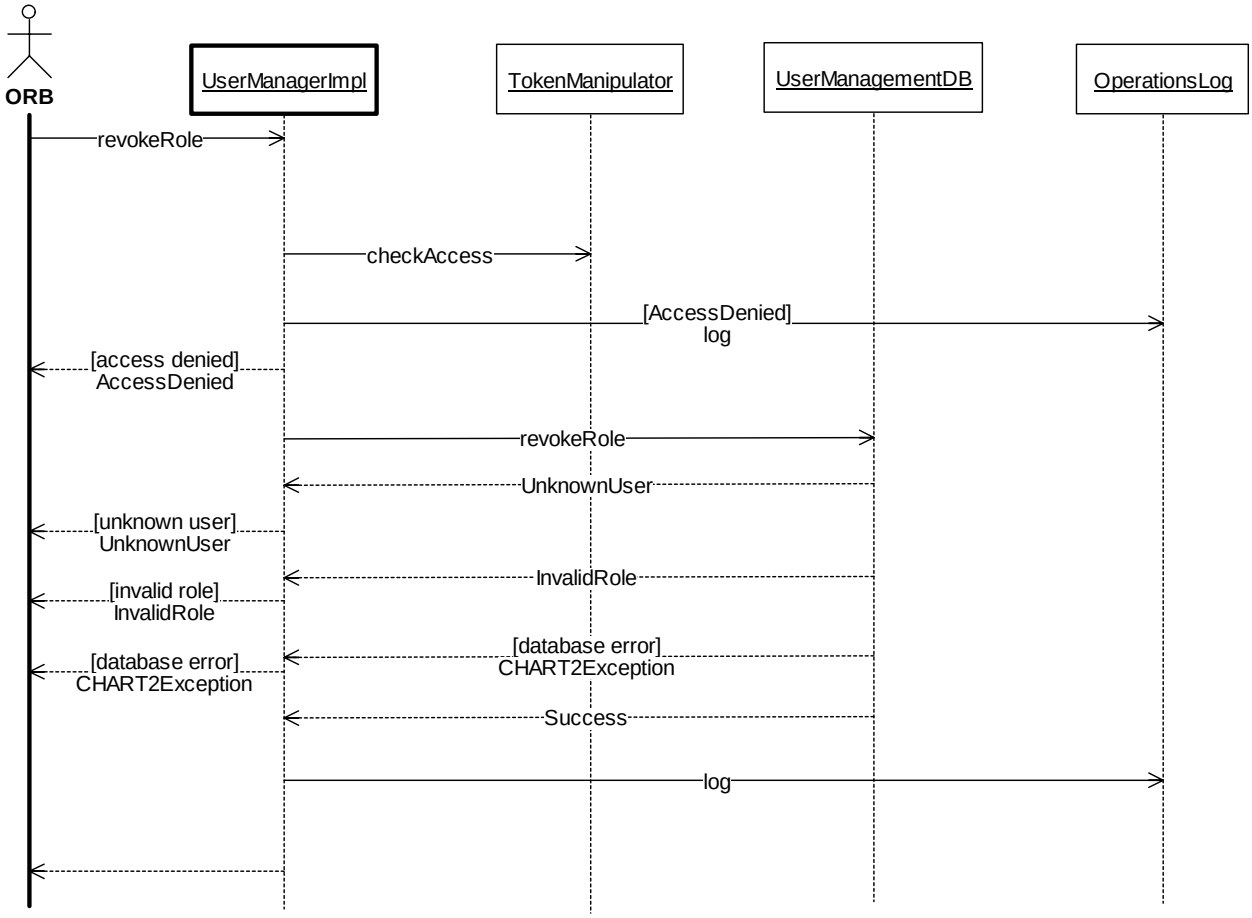

Figure 3-74. UserManagementModule:RevokeRole (Sequence Diagram)

3.8.2.10 UserManagementModule:SetUserPassword (Sequence Diagram)

A user with the proper functional rights may set the password that a user must specify in order to log into the system. This action does not require that the administrator be able to supply the users current password and, therefore, is restricted to administrative users. This function is included to deal with situations where users forget their system password.

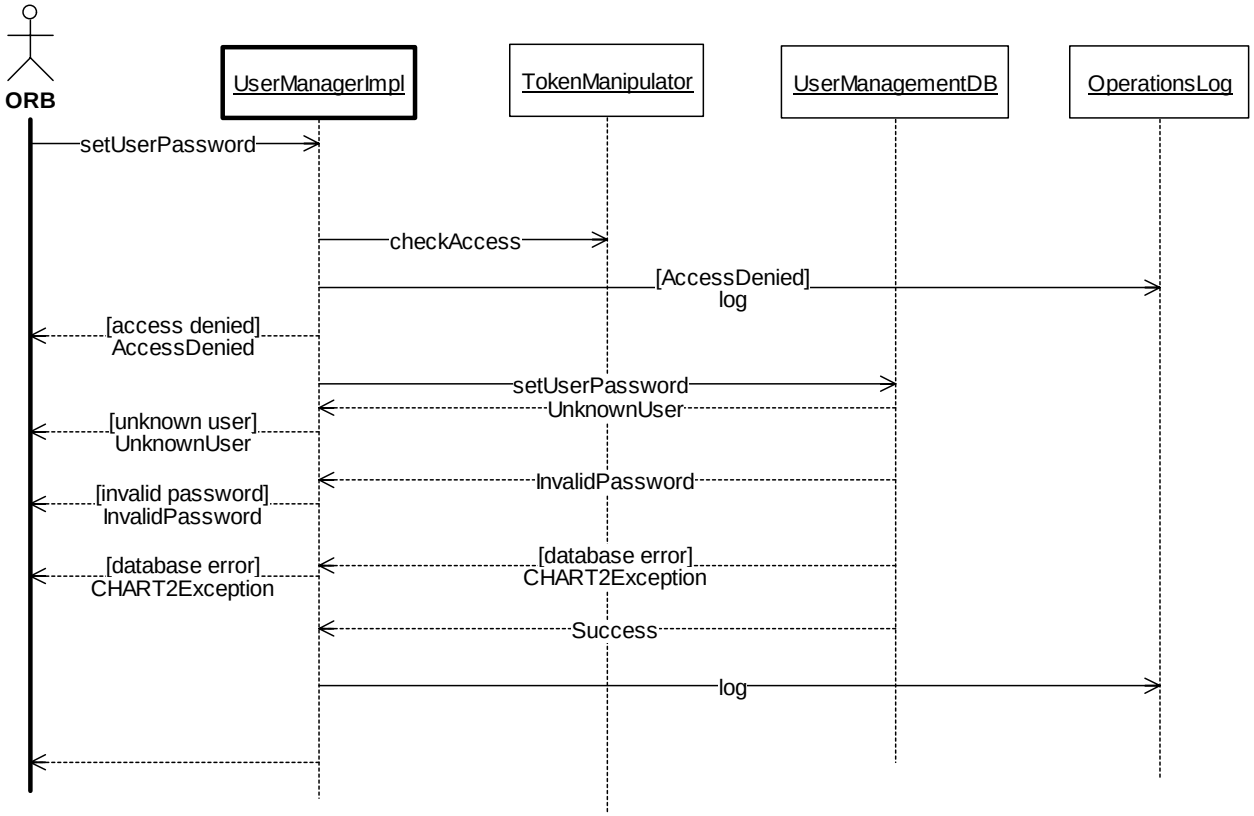

Figure 3-75. UserManagementModule:SetUserPassword (Sequence Diagram)

3.8.2.11 UserManagementModule:Shutdown (Sequence Diagram)

The user management module will withdraw the user management object from the trader, disconnect it from the ORB and delete it.

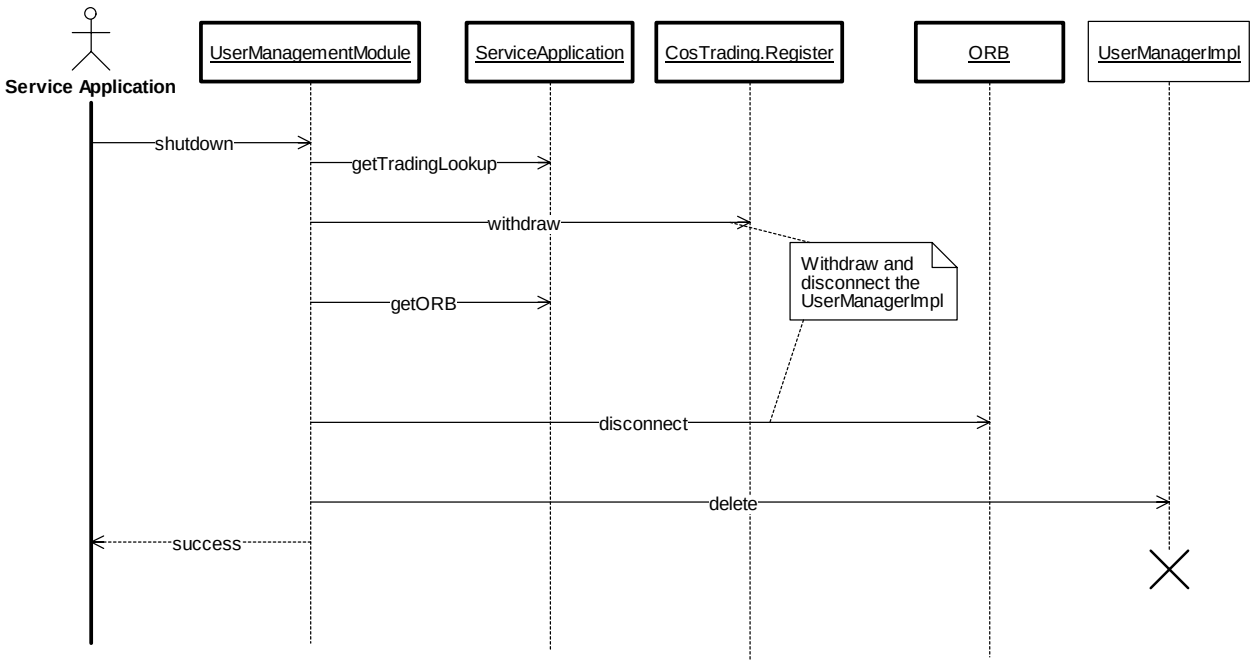

Figure 3-76. UserManagementModule:Shutdown (Sequence Diagram)

3.9 UserManagementResourcesModule

3.9.1 UserManagementResourceClasses (Class Diagram)

These classes represent the resource control portion of the User Management service application. These classes have been located in an installable service module in order to allow them to be served from an alternate service application in future releases of the system. The purpose of this module is to serve objects implementing the Organization interface and objects implementing the OperationsCenter interface.

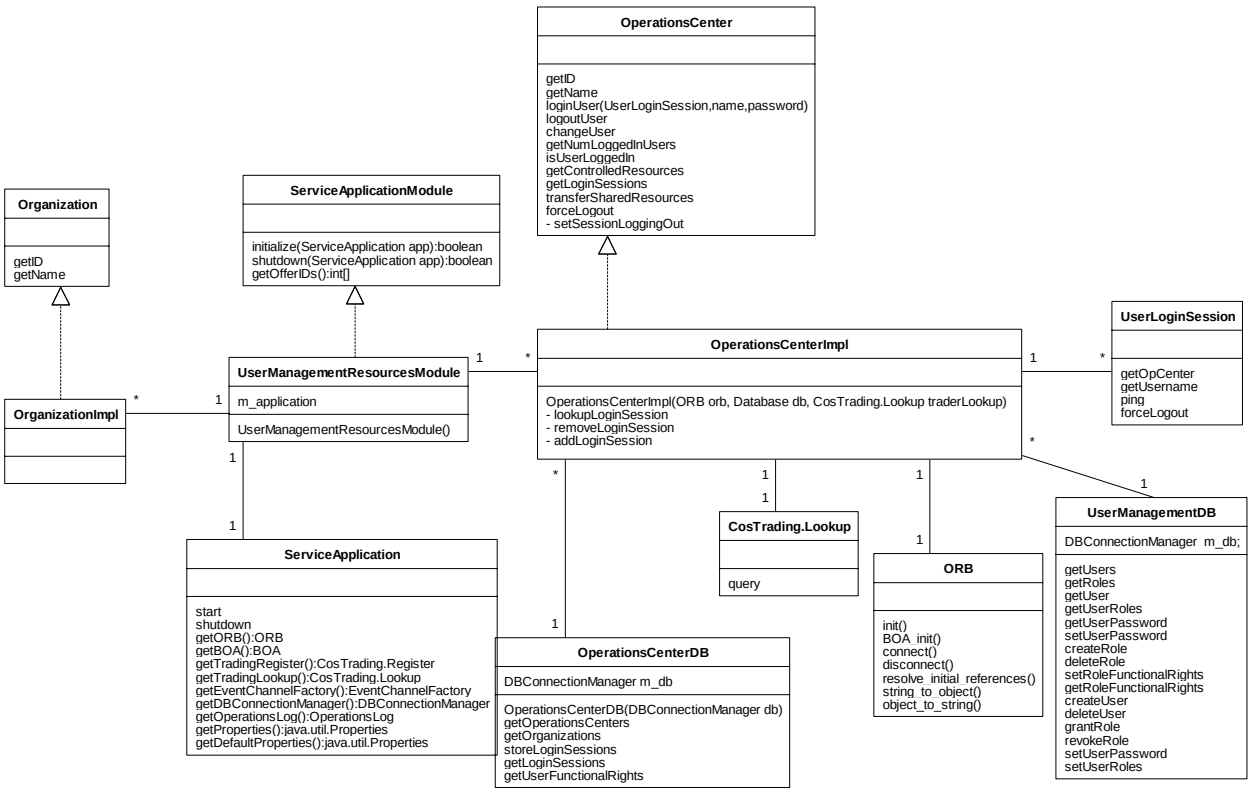

Figure 3-77. UserManagementResourceClasses (Class Diagram)

3.9.1.1 OperationsCenterDB (Class)

This class provides a set of API calls to access the Operations Center data from the database. The API's provide functionality to add, remove and retrieve Operation Center data from the database. The connection to the database is acquired from the Database object that manages all the database connections.

3.9.1.2 Organization (Class)

The Organization class represents an organization that participates in the Chart system through ownership of shared resources. The Organization can be used in conjunction with functional rights to determine the level of access users have to shared resources owned by a given organization. This allows access to be granted to a user to perform controlled operations on shared resources owned by one organization but not another.

1

interface

3.9.1.3 OrganizationImpl (Class)

This class provides the implementation of the Organization interface for this module. Thus, it provides a concrete implementation of each of the methods in the interface.

3.9.1.4 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.9.1.5 OperationsCenterImpl (Class)

This class provides the implementation of the OperationsCenter interface for this module. It, therefore, provides a concrete implementation of each of the methods in the interface. It also contains a collection of UserLoginSession objects, one for each user who is currently logged in.

3.9.1.6 UserLoginSession (Class)

The UserLoginSession class is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

interface

3.9.1.7 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

interface

3.9.1.8 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

1

interface

3.9.1.9 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

interface

3.9.1.10 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.9.1.11 UserManagementDB (Class)

The UserManagementDB Class provides methods used to access and modify User Managment data in the database. This class uses a Database object to retrieve a connection to the database for its exclusive use during a method call.

3.9.1.12 UserManagementResourcesModule (Class)

This module creates, publishes and destroys all objects related to resource management that are used by the User Management service application.

3.9.2 Sequence Diagrams

3.9.2.1 UserManagementResourcesModule:ChangeUser (Sequence Diagram)

A client with the correct functional rights may select to relinquish his/her workstation to another operator. This typically will happen at shift change. This sequence logs the new operator in before logging the old operator out. Thereby guaranteeing that the shared resources controlled by the operations center have a responsible operator during the transition. If this method throws any type of exception, the old user is still logged in and the new user is not. If this method returns a token, the old user is logged out and the new user is logged in.

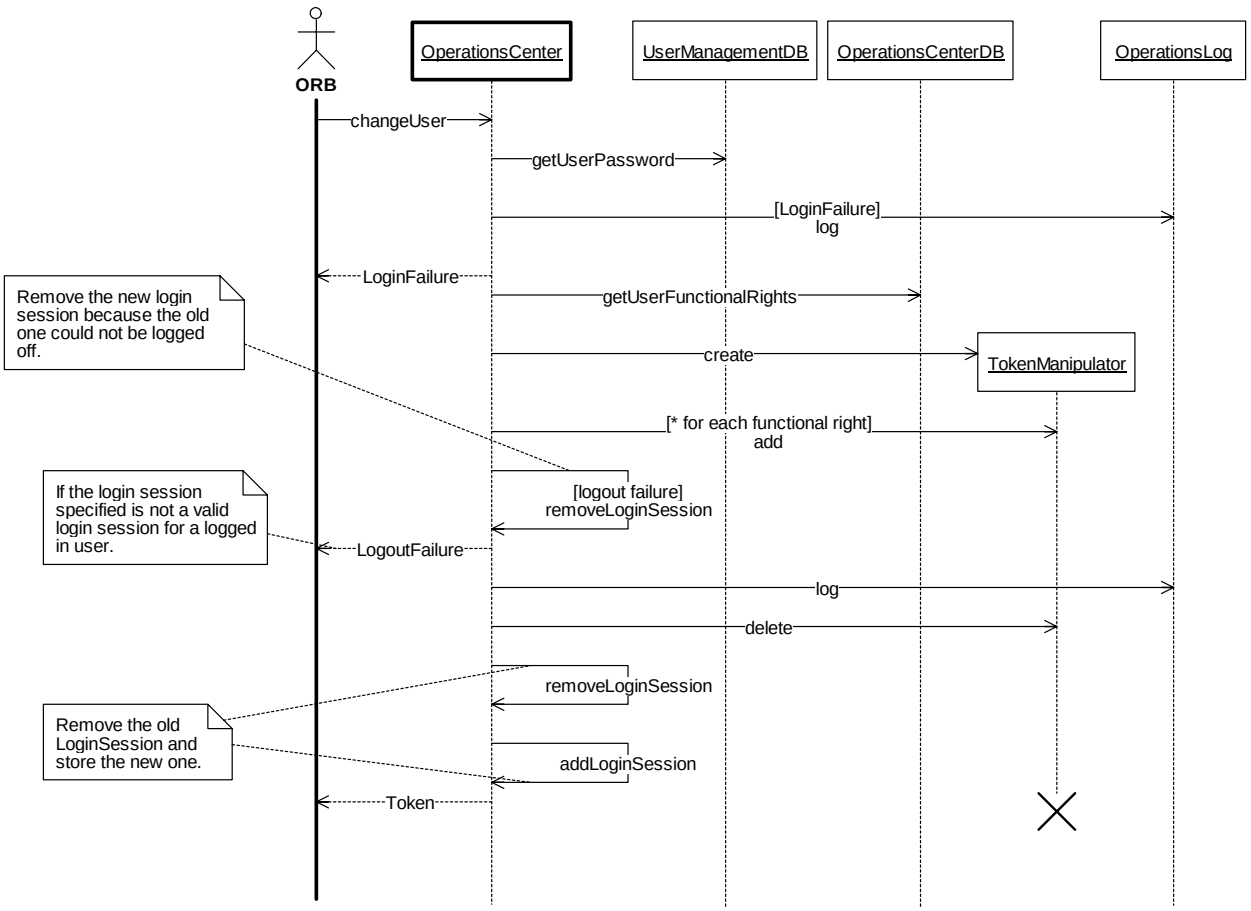

Figure 3-78. UserManagementResourcesModule:ChangeUser (Sequence Diagram)

3.9.2.2 UserManagementResourcesModule:ForceLogout (Sequence Diagram)

A client with the correct functional rights may force a particular user to logout of the CHART2 system. This is actually accomplished in two steps. The client would first need to acquire a UserLoginSession object before calling this method, please refer to the sequence diagram for the getUserLoginSessions method for details. Once the user has acquired a UserLoginSession he/she may contact the Operations Center where that UserLoginSession is being tracked and inform it that the user should be forced to logout. The OperationsCenter will call the forceLogout method on the specified UserLoginSession after removing the login session from its internal collection of login sessions. Note that it is possible for the user to call the forceLogout method directly on the UserLoginSession without informing the OperationsCenter. This method of forcing a user to logout is also accepted. If this path is taken, the operations center will contain a reference to a UserLoginSession which is no longer valid. This possibility is accounted for by pinging the UserLoginSession objects each time the getNumLoggedInUsers() method is called. Please refer to that sequence diagram for details.

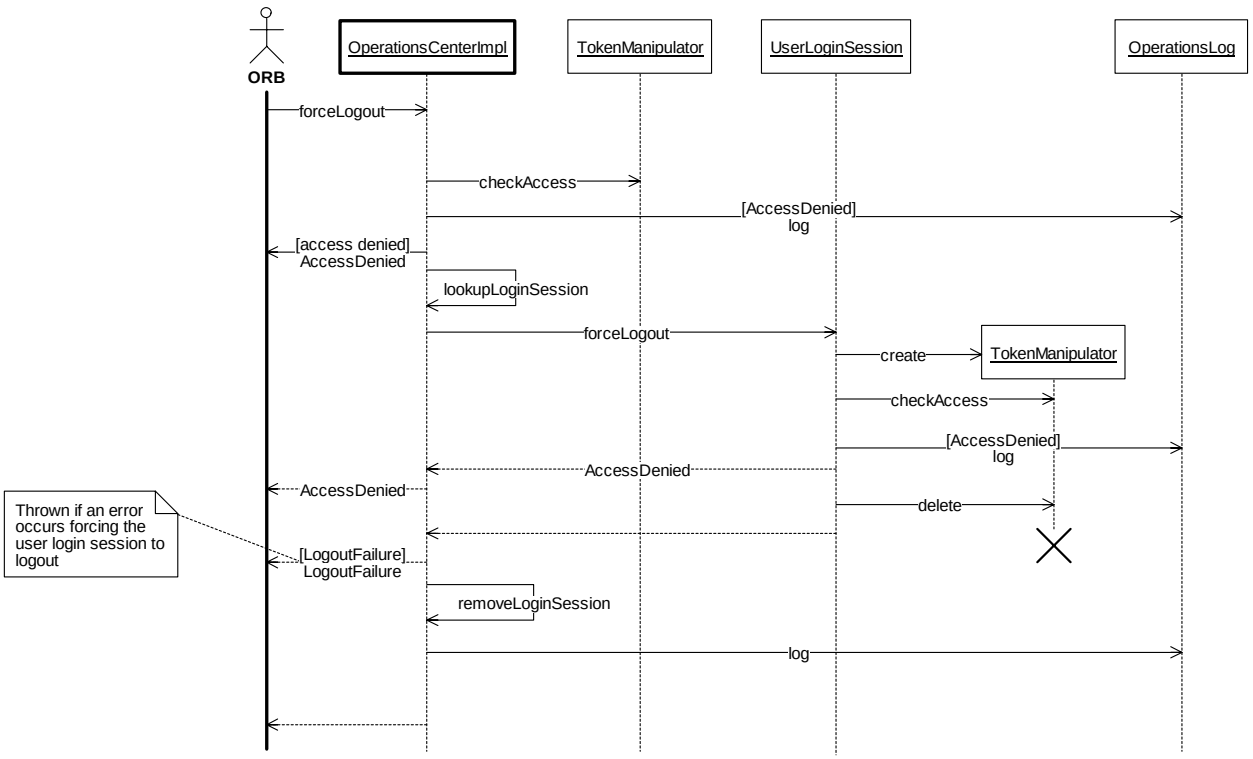

Figure 3-79. UserManagementResourcesModule:ForceLogout (Sequence Diagram)

3.9.2.3 UserManagementResourcesModule:GetControlledResources (Sequence Diagram)

A client may request a list of all shared resources that are currently controlled by this operations center. This would typically happen if the user were looking to transfer responsibility for some of all of the controlled shared resources from one operations center to another. The operations center will contact each shared resource manager in the system and get a list of resources which it is currently controlling. The lists returned by each shared resource manager will be combined and the entire list of controlled resources will be returned to the user.

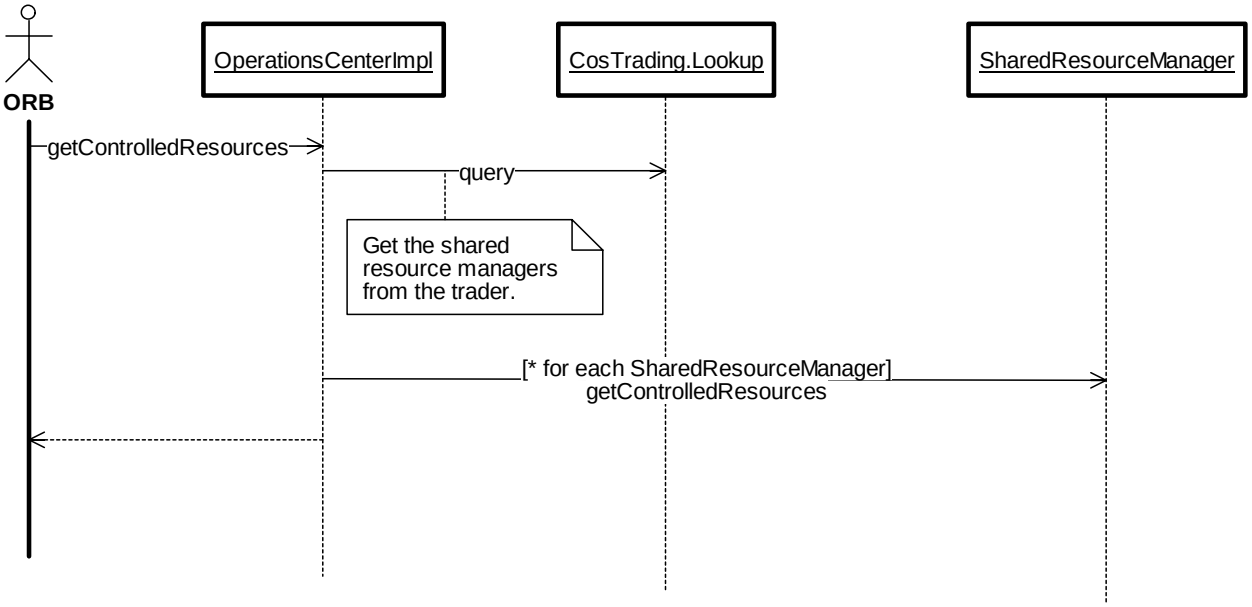

Figure 3-80. UserManagementResourcesModule:GetControlledResources (Sequence Diagram)

3.9.2.4 UserManagementResourcesModule:GetLoginSessions (Sequence Diagram)

A client with the correct functional rights may get a list of UserLoginSessions that represents the list of users who are currently logged in from this operations center.

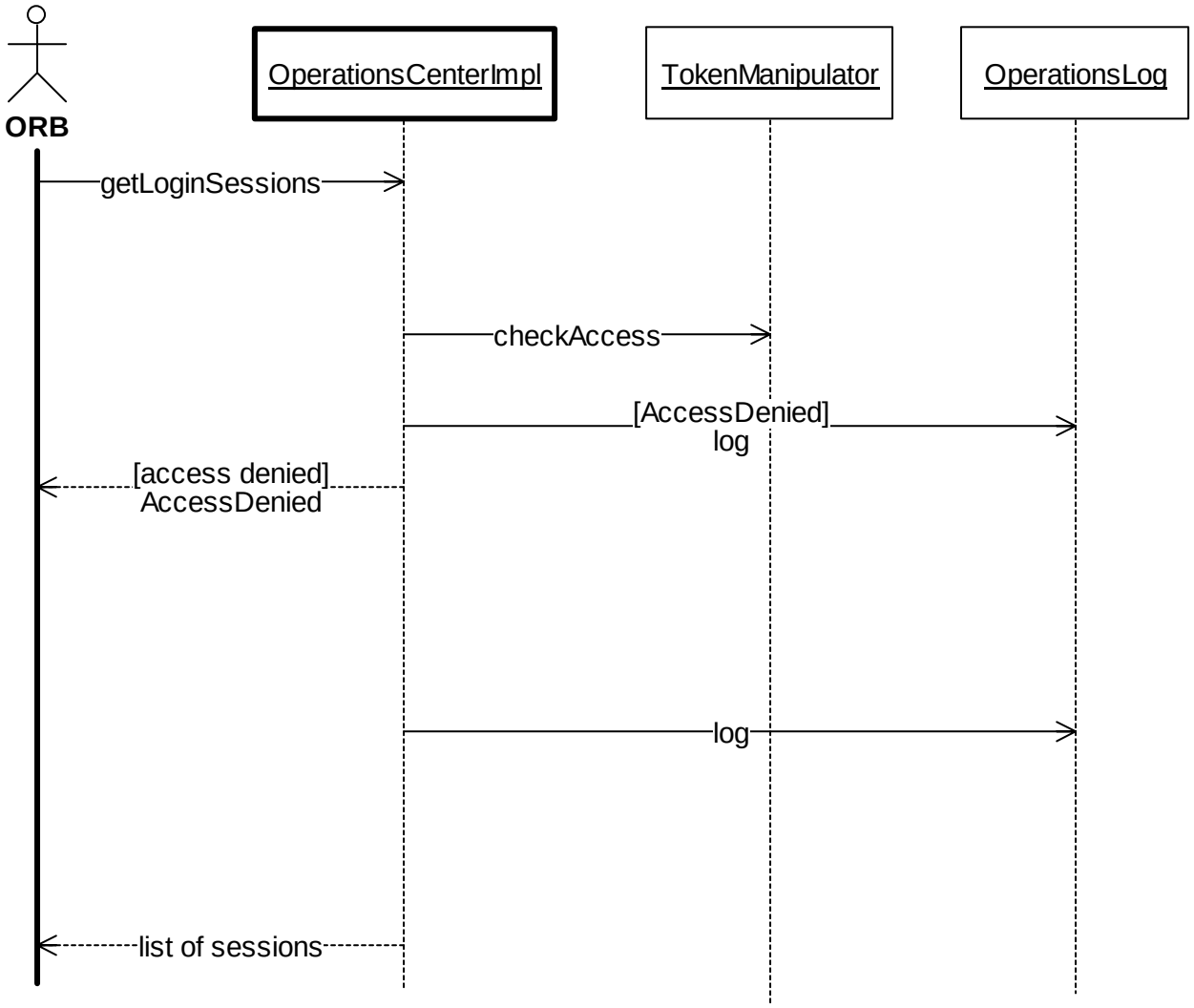

Figure 3-81. UserManagementResourcesModule:GetLoginSessions (Sequence Diagram)

3.9.2.5 UserManagementResourcesModule:GetNumLoggedInUsers (Sequence Diagram)

This method allows a client to get the number of users who are currently logged in at this operations center. This method will be used by the shared resource manager watchdogs to verify that they do not have shared resources which are under the control of operations centers with no users logged in. This method will ping each UserLoginSession before counting it as a valid login session. The ping protects the system from counting login sessions from GUI's which have been turned off or disconnected without performing a proper logout.

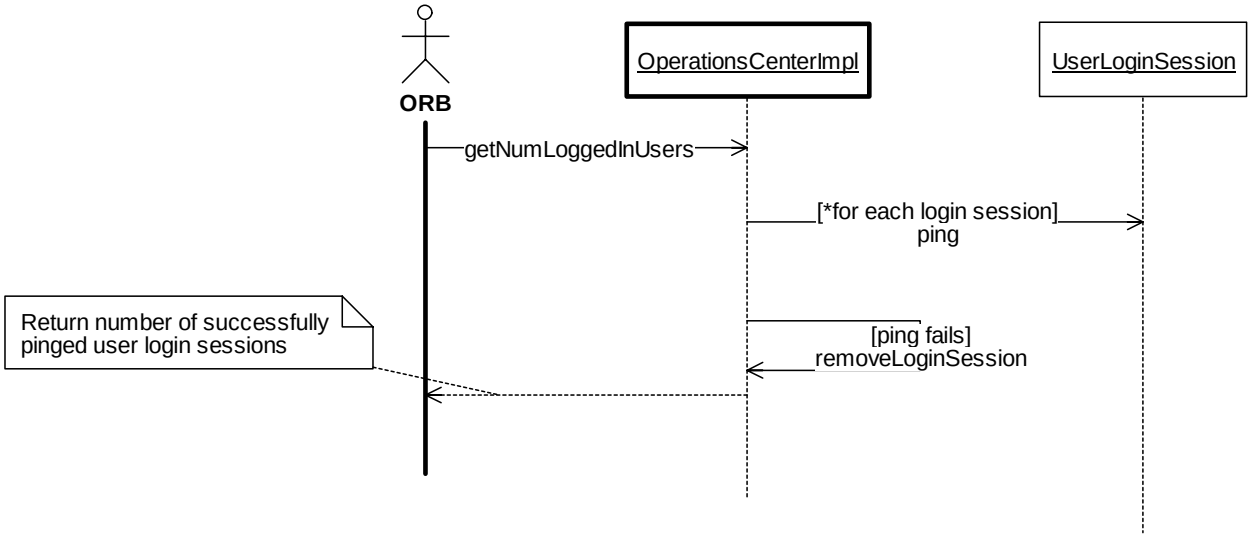

Figure 3-82. UserManagementResourcesModule:GetNumLoggedInUsers (Sequence Diagram)

3.9.2.6 UserManagementResourcesModule:IsUserLoggedIn (Sequence Diagram)

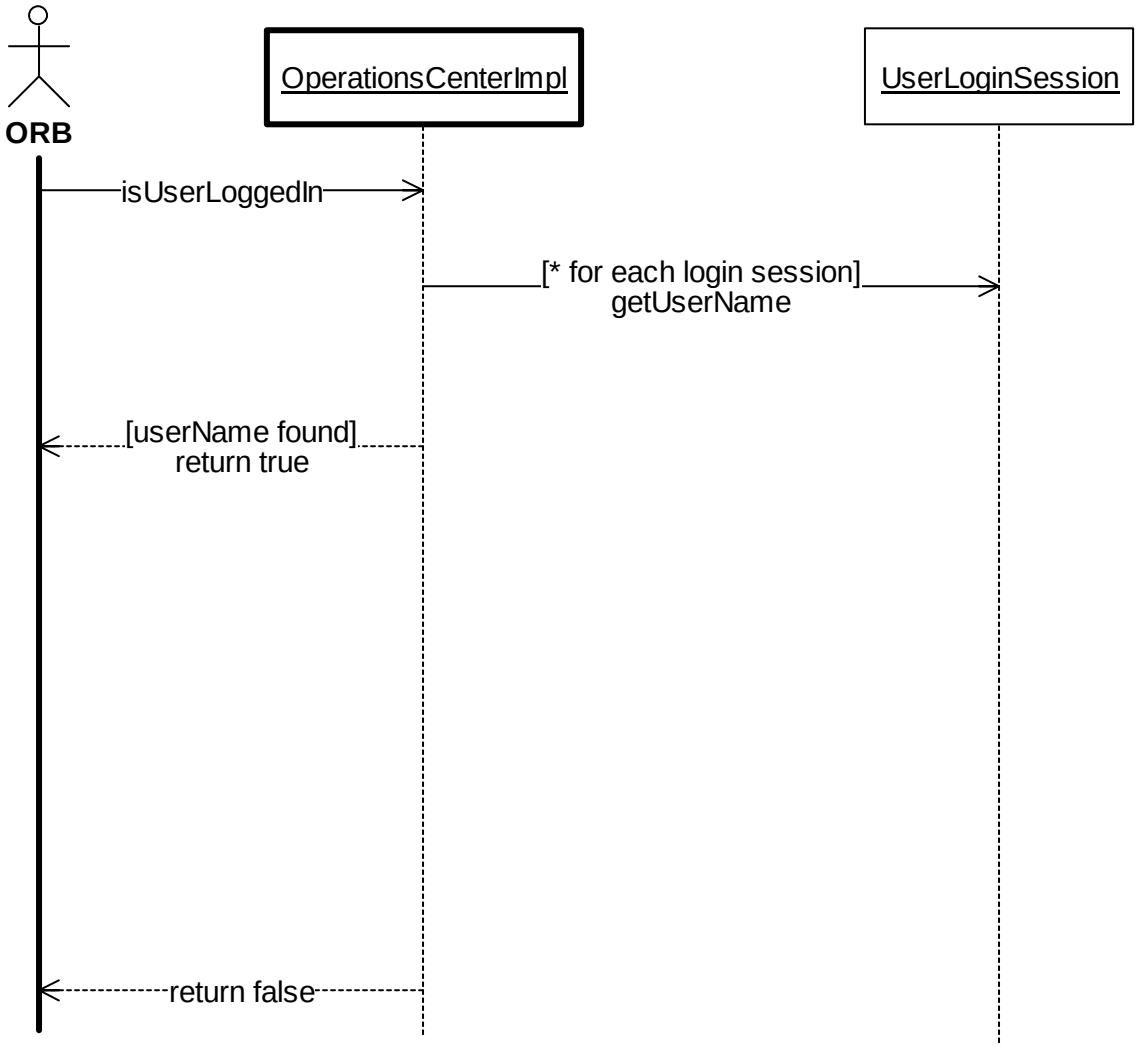

Figure 3-83. UserManagementResourcesModule:IsUserLoggedIn (Sequence Diagram)

3.9.2.7 UserManagementResourcesModule:LoginUser (Sequence Diagram)

A client may login to the system. The system will verify that the user has specified the correct password by looking in the user database. If the user has specified the correct password, the system will create a token that contains the user's functional rights and will return it to the invoking client. The login session will be stored internally in the operations center in order to allow the center to respond to calls regarding shared resource control.

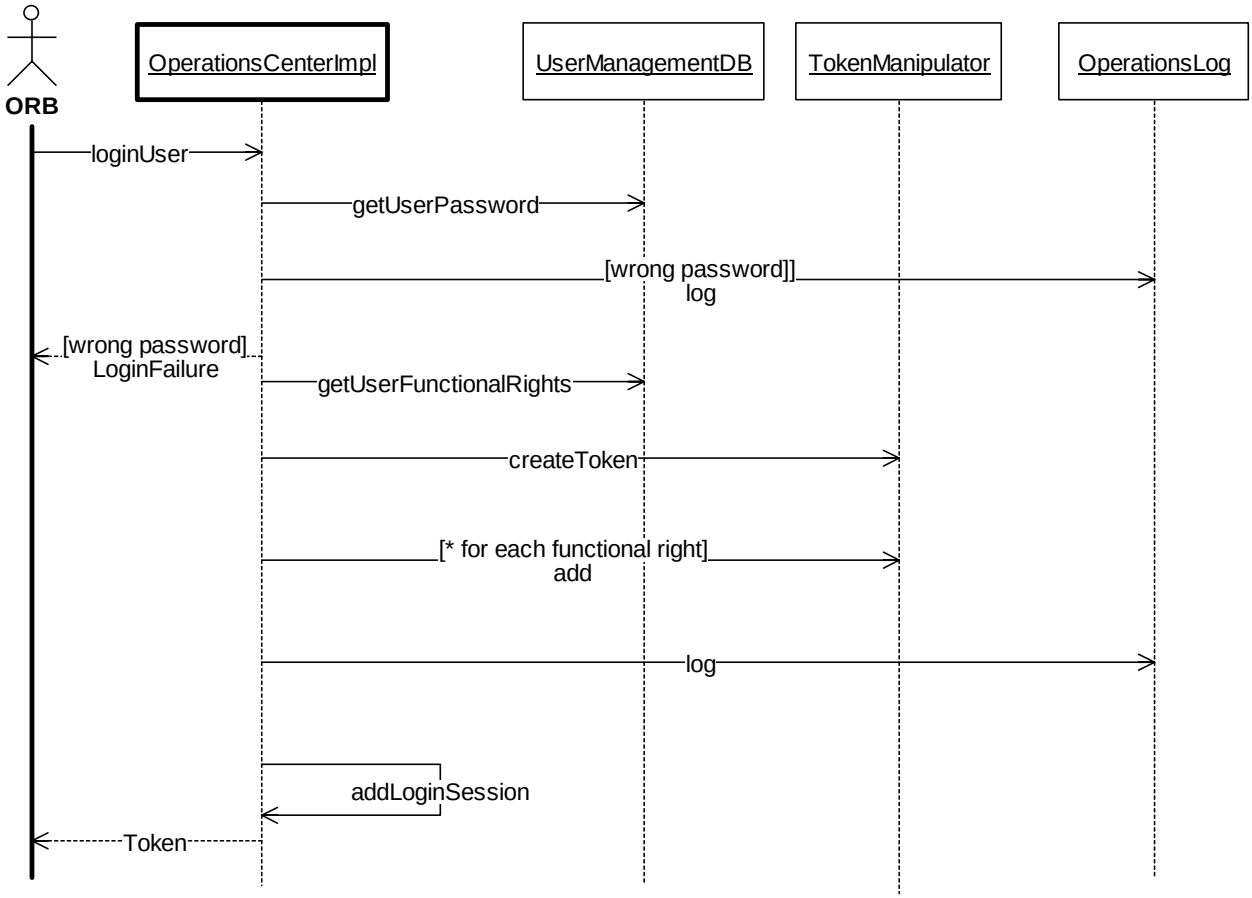

Figure 3-84. UserManagementResourcesModule:LoginUser (Sequence Diagram)

3.9.2.8 UserManagementResourcesModule:LogoutUser (Sequence Diagram)

A client may log out of the system. When an operator does this, the system will ping each user login session it is tracking to verify the actual number of users who are currently logged in. If the current number of valid login sessions for this operations center is one, then this user cannot be allowed to logout if this operations center is currently controlling shared resources. In order to determine if the operations center has controlled resources, the system will contact all of the shared resource managers. If the operations center has controlled resources an exception will be thrown, otherwise the user will be logged out.

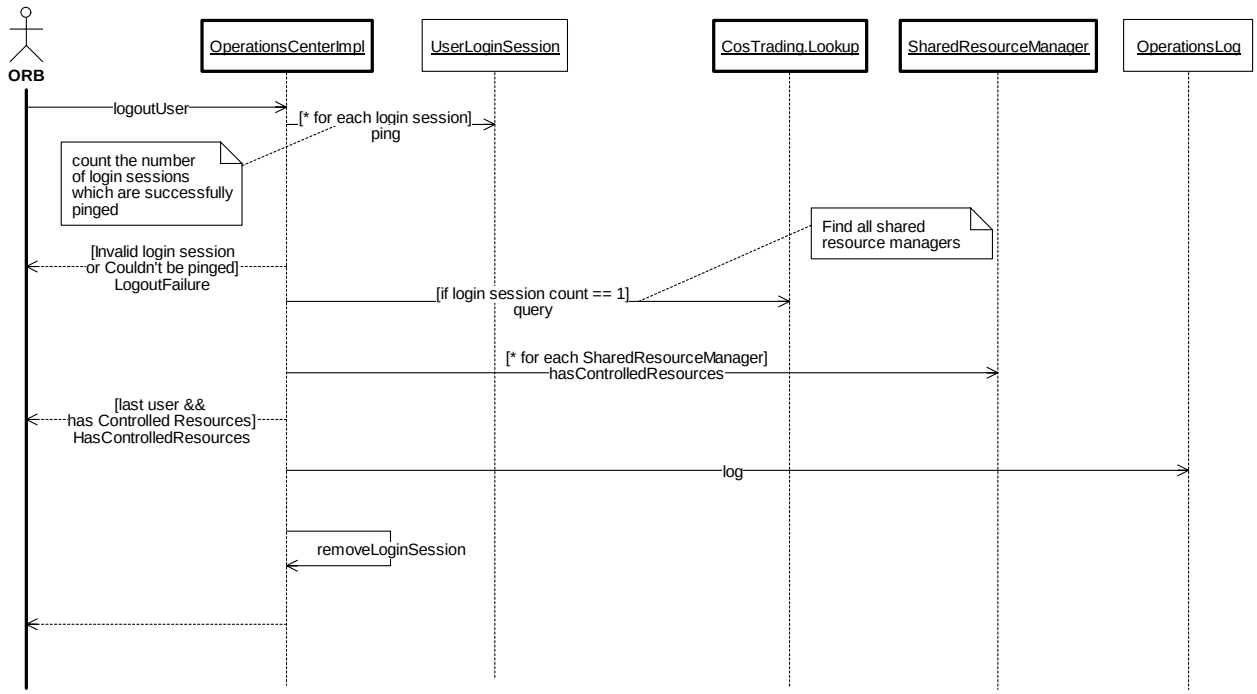

Figure 3-85. UserManagementResourcesModule:LogoutUser (Sequence Diagram)

3.9.2.9 UserManagementResourcesModule:OperationsCenterImplInitialization (Sequence Diagram)

This sequence shows the details of constructing an operations center implementation object. An operations center is responsible for tracking the list of currently logged in users. When the service is shutdown it will store the list in the database. When the service is restarted it will get this list of login sessions from the database. Because the service may have been down for an extended period, the login sessions may no longer be valid due to users logging out or shutting down their client machines. Thus, each login session object will be pinged to see if it is still active. If it is, the operations center will add it to the list of current sessions otherwise it will not.

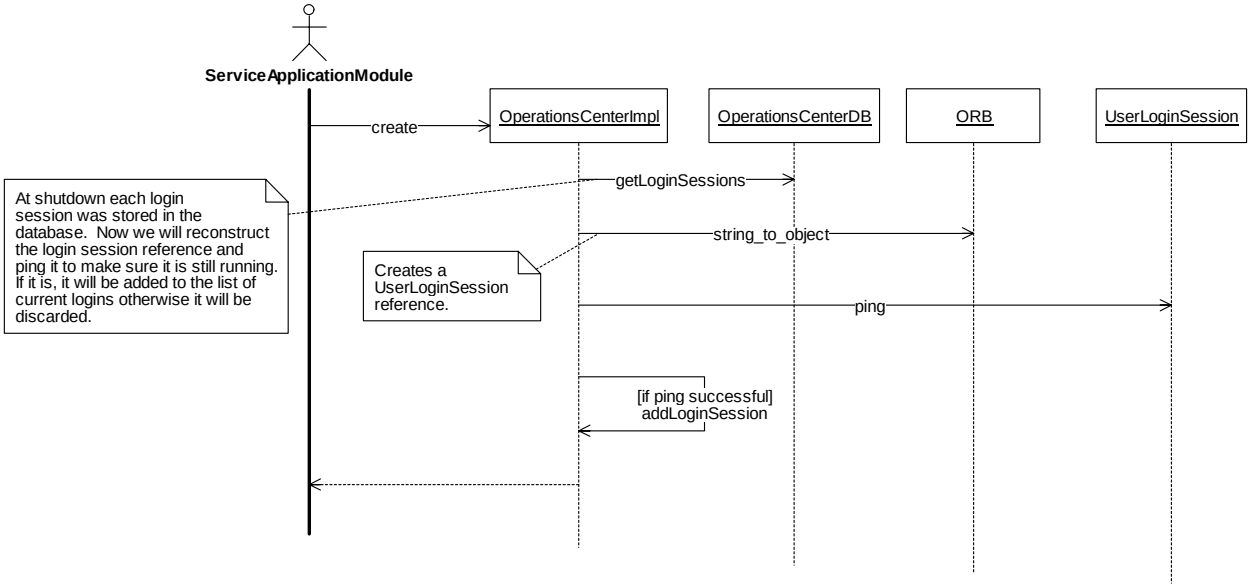

Figure 3-86. UserManagementResourcesModule:OperationsCenterImplInitialization (Sequence Diagram)

3.9.2.10 UserManagementResourcesModule:Shutdown (Sequence Diagram)

When the service application calls the shutdown method on this module, the module will withdraw all exported offers from the trader, disconnect any objects that it is currently serving from the ORB and destroy them. The operations center will also store the current list of UserLoginSession references in the database. This will allow the login sessions to be reconstructed at startup.

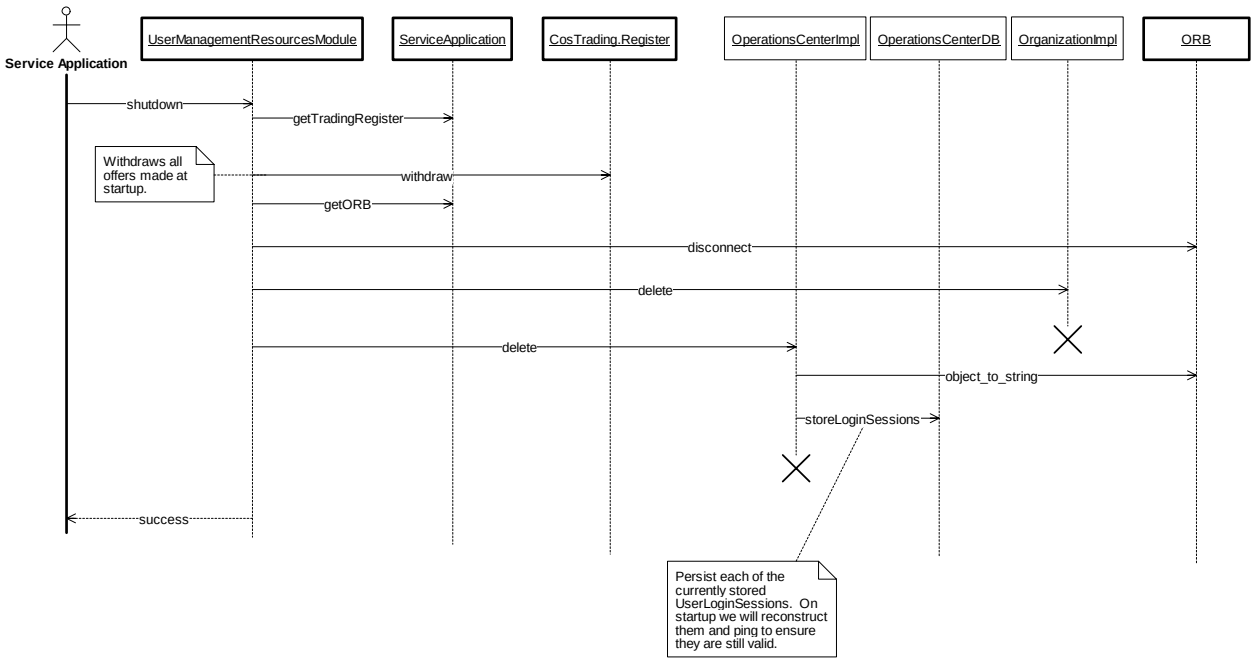

Figure 3-87. UserManagementResourcesModule:Shutdown (Sequence Diagram)

3.9.2.11 UserManagementResourcesModule:Initialize (Sequence Diagram)

When the service is started, the service application will call initialize on this module. The module will create the operations center and organization implementation objects which are found in the database, connect them to the ORB and export them in the trading service so that other applications may locate them.

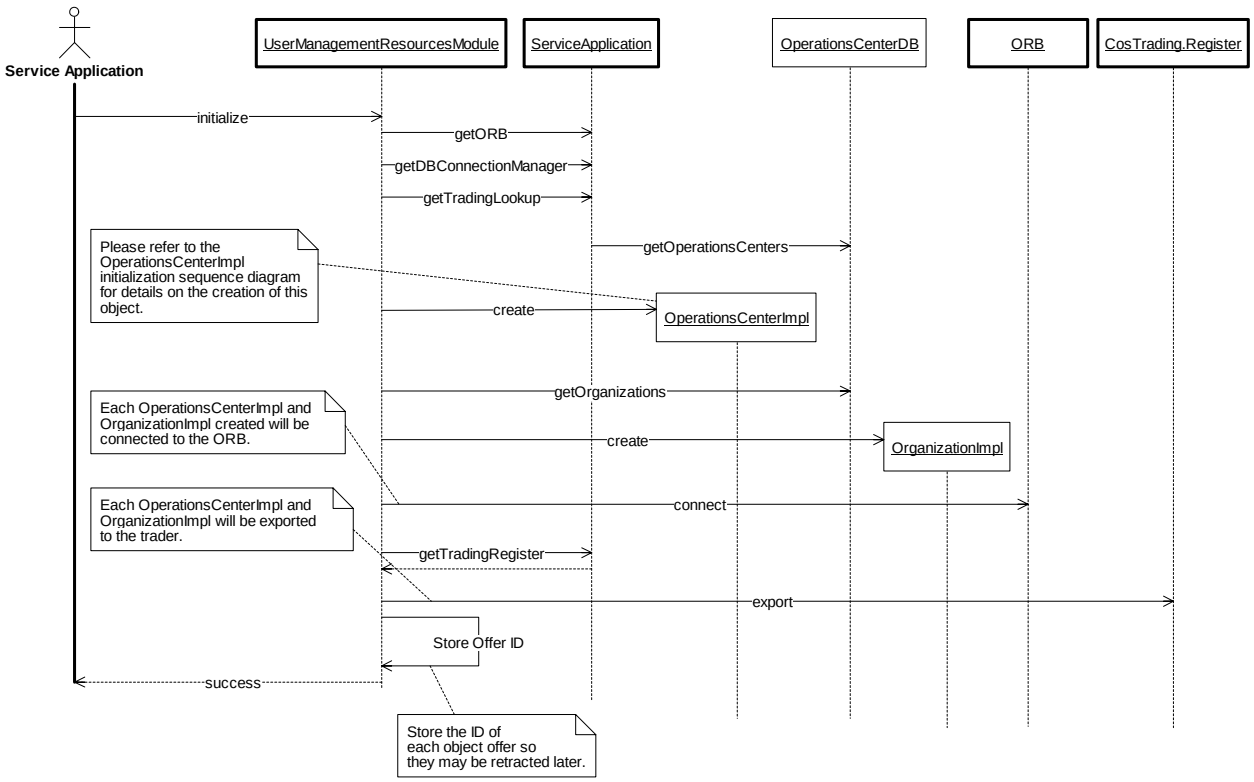

Figure 3-88. UserManagementResourcesModule:Initialize (Sequence Diagram)

3.9.2.12 UserManagementResourcesModule:TransferSharedResources (Sequence Diagram)

A client with the correct functional rights may transfer the control of shared resources from this operations center to another. The system will verify that there are users logged in at the target operations center and will then transfer control of the shared resources if there are.

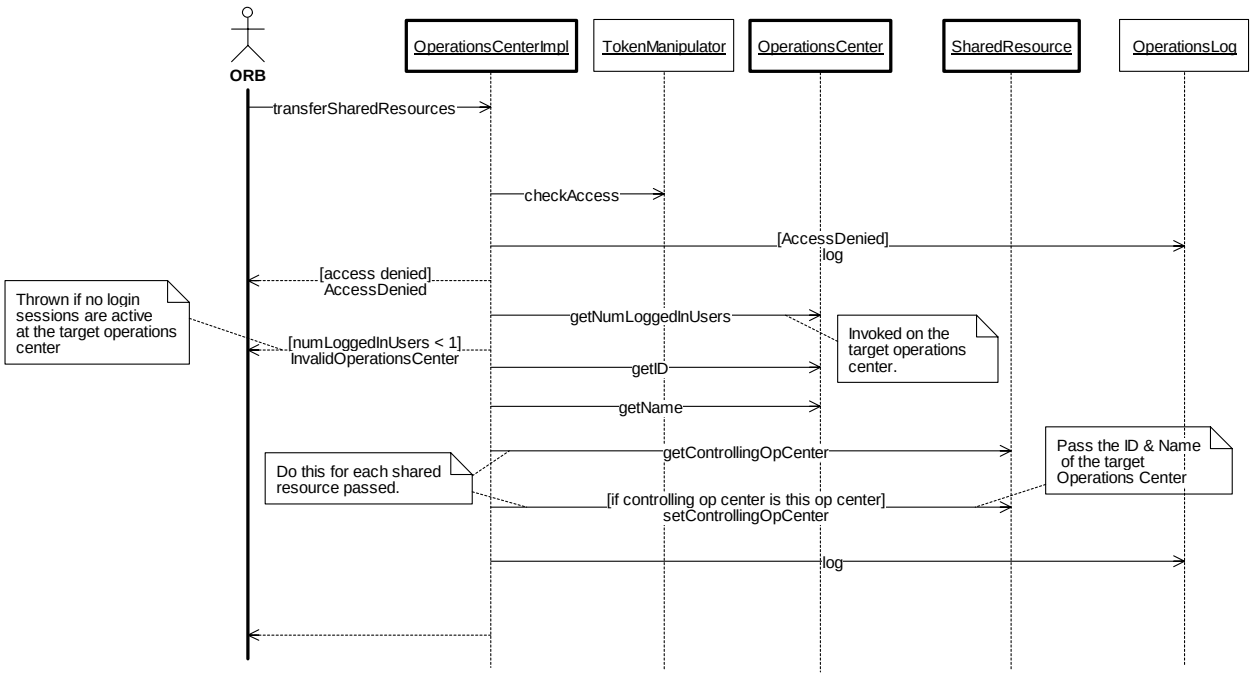

Figure 3-89. UserManagementResourcesModule:TransferSharedResources (Sequence Diagram)

3.10 ExtendedEventService

3.10.1 ExtendedEventServiceClasses (Class Diagram)

The ExtendedEventService is an extension to the standard CORBA Event Service that allows multiple channels to be created within the event service. The EventChannelFactory CORBA interface is provided to allow others to create and remove standard CORBA event channel objects.

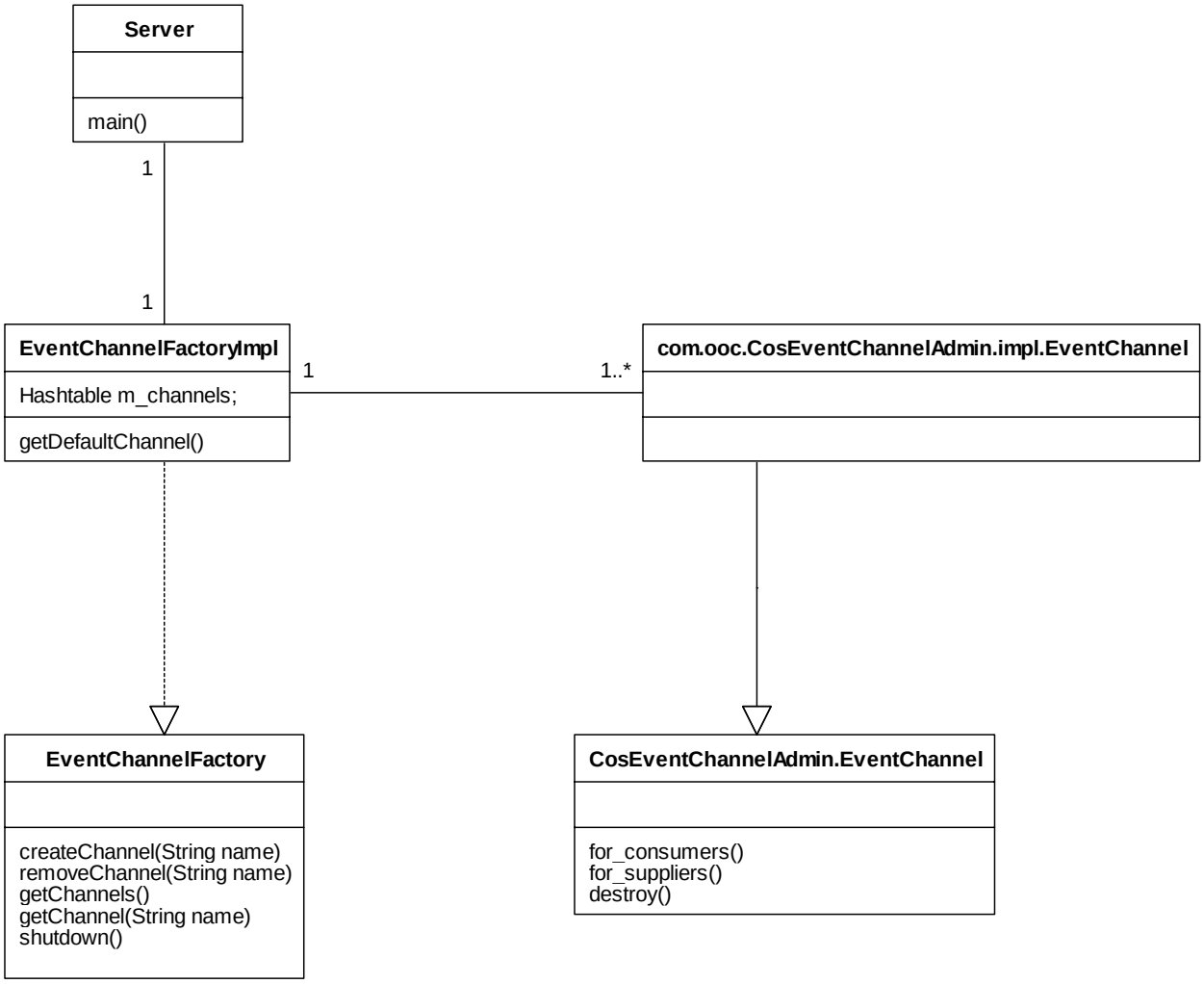

Figure 3-90. ExtendedEventServiceClasses (Class Diagram)

3.10.1.1 com.ooc.CosEventChannelAdmin.impl.EventChannel (Class)

This class is the ORB vendor's implementation of a CORBA event channel. The event service provided by the vendor simply serves one of these objects. The Extended Event Service serves a factory that allows multiple instances of the vendor supplied event channel to be created.

3.10.1.2 CosEventChannelAdmin.EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

interface

3.10.1.3 EventChannelFactory (Class)

This interface defines the operations used to provide a collection of event channels that are served through a CORBA ORB.

interface

3.10.1.4 EventChannelFactoryImpl (Class)

This class implements the EventChannelFactory interface that is defined by IDL. It provides methods to allow for the creation and removal of standard corba event channel objects. The implemenation of the event channel interface is provided by the ORB vendor.

3.10.1.5 Server (Class)

This class implements a main method that is used to serve an EventChannelFactory and the Event Channels that are managed by the factory. This class is used to run the ExtendedEventService, which extends the standard CORBA event service by allowing multiple channels to be served from the same event service.

3.10.2 Sequence Diagrams

3.10.2.1 ExtendedEventService:CreateChannel (Sequence Diagram)

When the factory is requested to add a new channel, the factory first checks to see if a channel with the given name already exists. If it does, an exception is thrown, returning a reference to the existing channel. Otherwise, a new channel is created, connected to the ORB, and stored in the factory. The new event channel is then passed back to the caller.

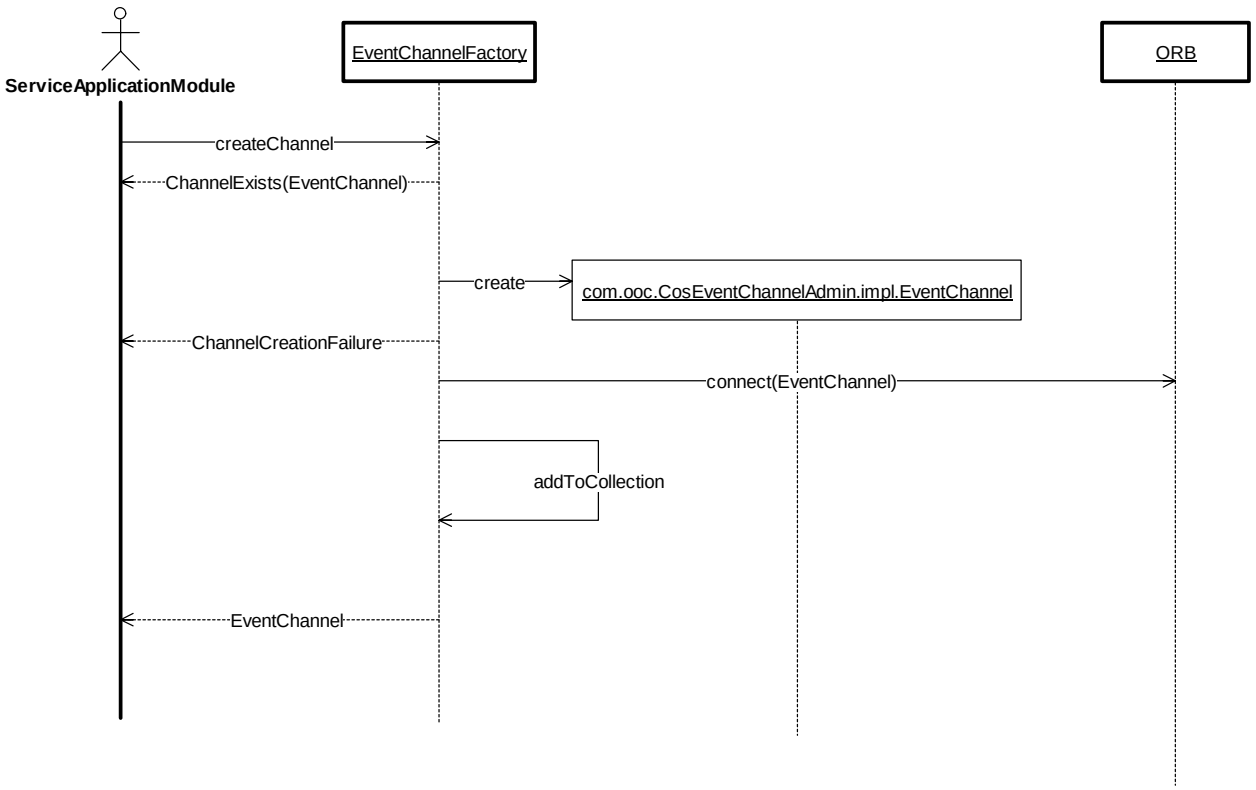

Figure 3-91. ExtendedEventService:CreateChannel (Sequence Diagram)

3.10.2.2 ExtendedEventService:Startup (Sequence Diagram)

During startup of the Extended Event Service, the main routine must initialize the ORB and create an instance of the EventChannelFactoryImpl. The EventChannelFactoryImpl creates a default Event channel and connects it to the ORB. This keeps the ExtendedEventService providing the same functionality as the standard event service, thus a user that does not wish to take advantages of the extended capabilities can use the service in a standard way. The EventChannelFactory is connected to the ORB to provide the extended capabilities.

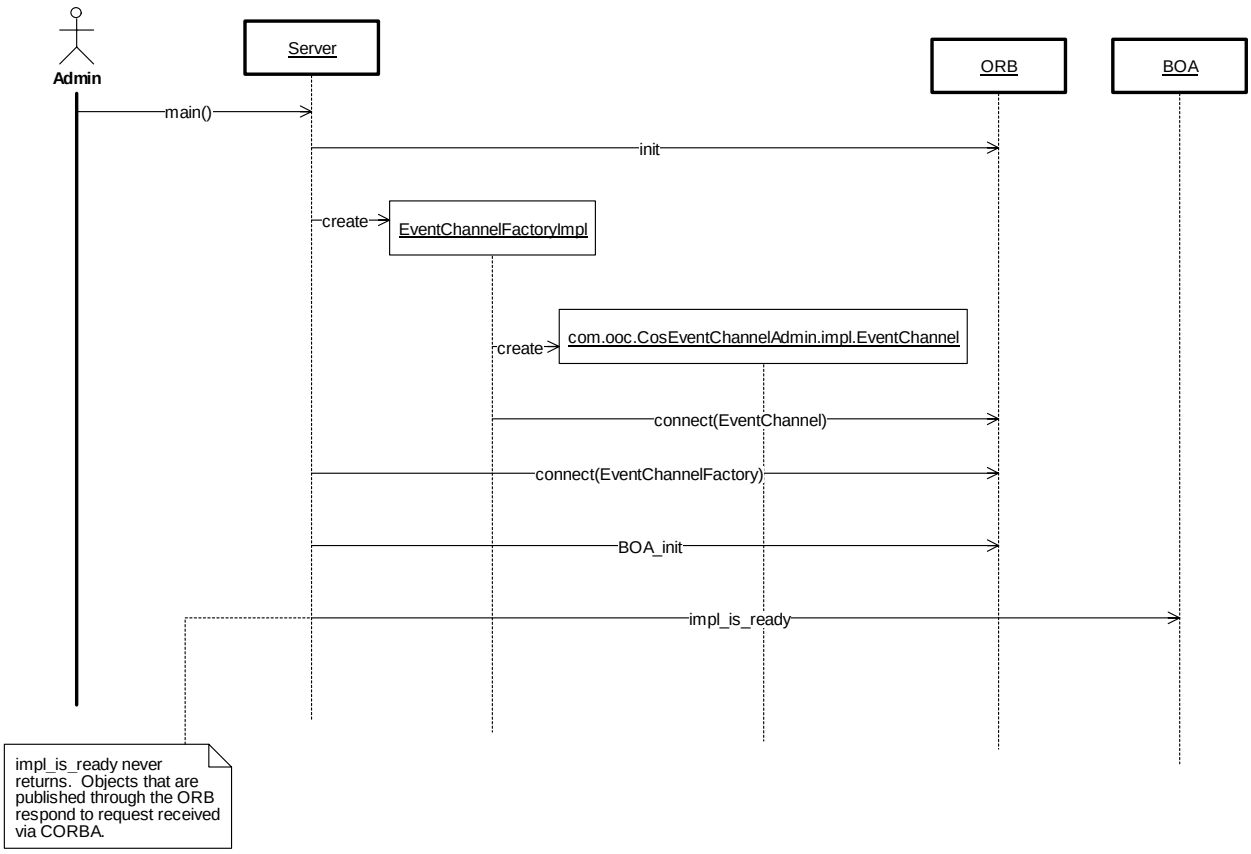

Figure 3-92. ExtendedEventService:Startup (Sequence Diagram)

3.11 System Interfaces

3.11.1 SystemInterfaces (Class Diagram)

This class diagram shows the interfaces from the High Level Design that are defined using IDL. These interfaces are included as reference and are included on other class diagrams in this design.

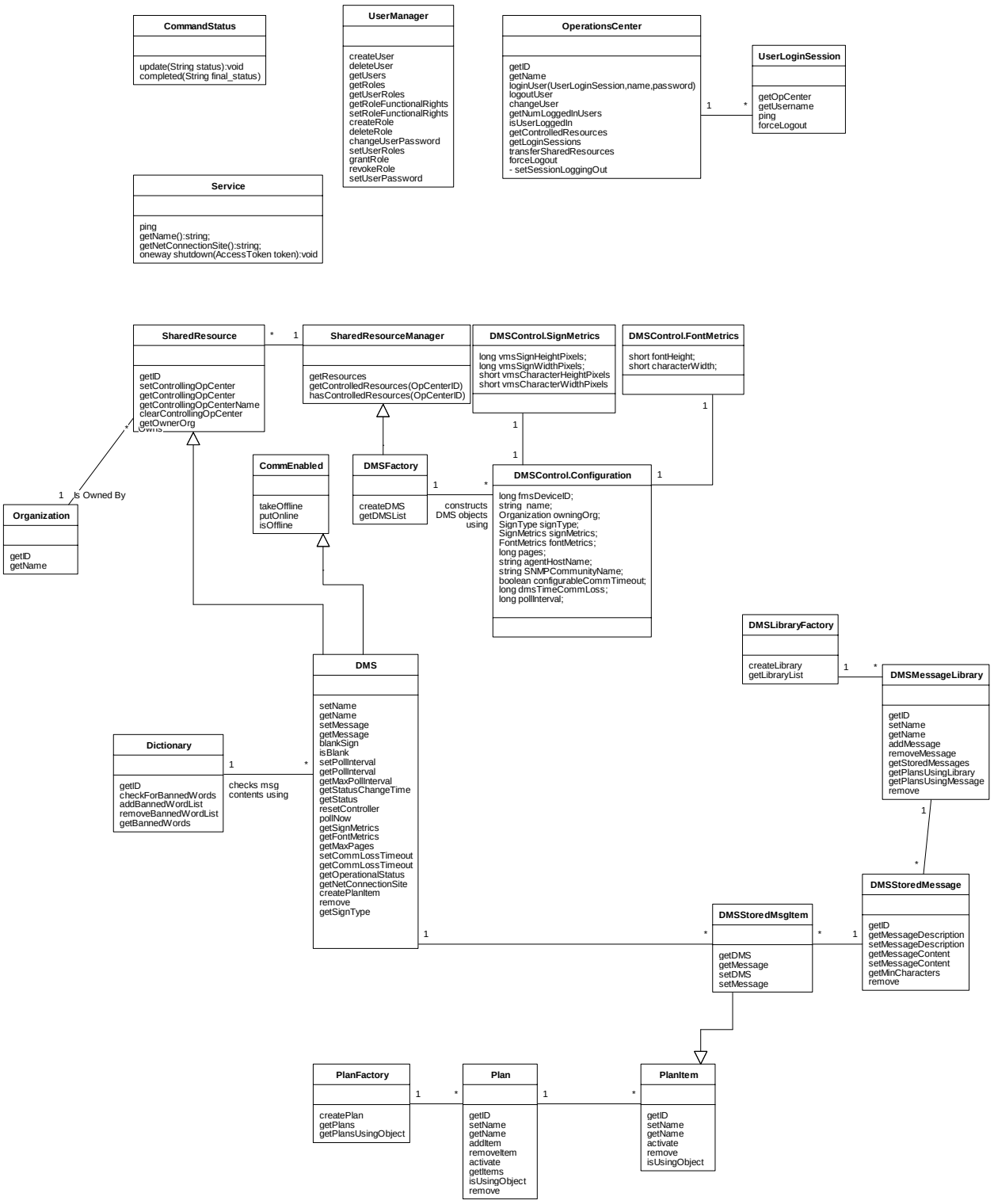

Figure 3-93. SystemInterfaces (Class Diagram)

3.11.1.1 CommandStatus (Class)

The CommandStatus class is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

interface

3.11.1.2 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can have their communications turned on or off. This typically only applies to field devices.

1

interface

3.11.1.3 Dictionary (Class)

This class is used to check for banned words in a message that may be displayed on a DMS. In addition to methods for checking the words, it has methods to allow the contents of the dictionary to be changed.

interface

3.11.1.4 DMS (Class)

This class represents a Dynamic Message Sign (DMS). It has attributes and methods for controlling and maintaining the status of the DMS within the system.

interface

3.11.1.5 DMSFactory (Class)

The DMSFactory provides a means to create new DMS objects to be added to the system.

1

interface

3.11.1.6 DMSMessageLibrary (Class)

This class represents a logical collection of stored DMS messages which are stored in the database.

interface

3.11.1.7 DMSStoredMessage (Class)

This class represents a stored DMS message that is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

interface

3.11.1.8 DMSStoredMsgItem (Class)

This class represents a plan item that is used to associate a stored DMS message with a specific DMS. When the item is activated, it sets the message of the DMS to the stored message to which it is linked.

interface

3.11.1.9 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

1

interface

3.11.1.10 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center.

1

interface

3.11.1.11 Organization (Class)

The Organization class represents an organization that participates in the Chart system through ownership of shared resources. The Organization can be used in conjunction with functional rights to determine the level of access users have to shared resources owned by a given organization. This allows access to be granted to a user to perform controlled operations on shared resources owned by one organization but not another.

1

interface

3.11.1.12 UserLoginSession (Class)

The UserLoginSession class is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

interface

3.11.1.13 DMSControl.Configuration (Class)

This typedef defines data that is used to identify the configuration of a DMS in the system.

typedef

3.11.1.14 DMSControl.FontMetrics (Class)

This typedef is included in the IDL to specify the data to be passed to/from operations to initialize or query the size of the font used by a DMS.

typedef

3.11.1.15 DMSControl.SignMetrics (Class)

This typedef is included in the IDL to specify the data included in operations that initialize or query the size of a DMS.

typedef

3.11.1.16 Plan (Class)

This class has a collection of Plan Items which it maintains. It has functionality for changing the plan items, and also allows the plan to be activated, which has the effect of activating each plan item in the plan.

interface

3.11.1.17 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans which can be used in the system.

interface

3.11.1.18 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

interface

3.11.1.19 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

interface

3.11.1.20 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

1

interface

3.11.1.21 DMSLibraryFactory (Class)

This class is used to create new DMS libraries and maintain them in a collection.

interface

3.11.1.22 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

1

interface

3.12 Utility

3.12.1 UtilityClasses (Class Diagram)

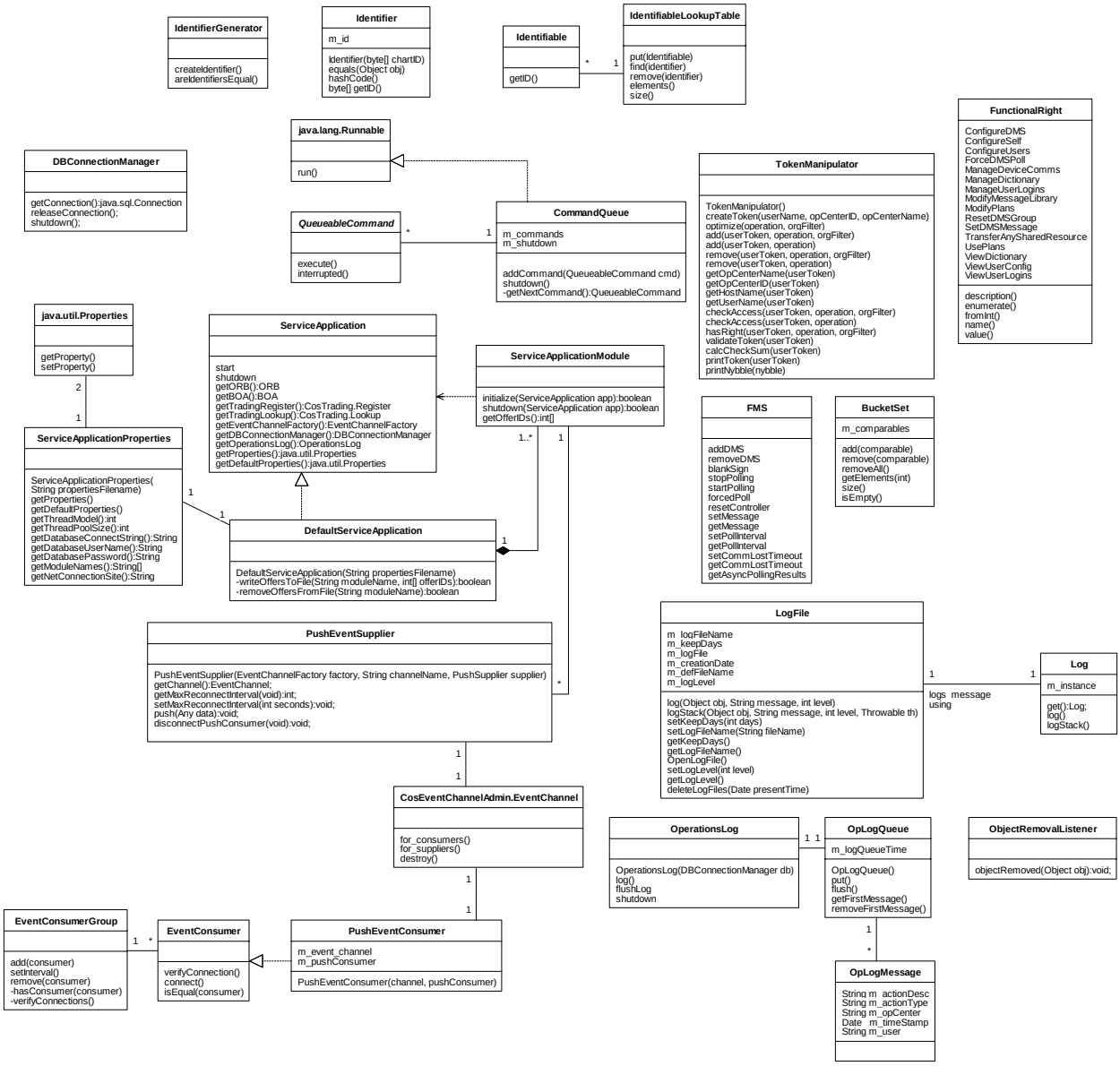

Figure 3-94. UtilityClasses (Class Diagram)

3.12.1.1 BucketSet (Class)

This class is designed to contain a collection of comparable objects. All of the objects added to this collection must be of the same concrete type. Each element in the collection has an associated counter that tracks how many times this element has been added. It is then possible to get only the elements which have been added to the collection n times where n is a positive integer value. This class is very useful for creating GUI menu's for multiple objects as it allows all objects to insert their menu items and then allows the user to get only those items with all objects inserted.

3.12.1.2 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

3.12.1.3 CosEventChannelAdmin.EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

interface

3.12.1.4 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependent to receive events.

3.12.1.5 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

interface

3.12.1.6 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults;" this second property list is searched if the property key is not found in the original property list.

3.12.1.7 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the

DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

3.12.1.8 OperationsLog (Class)

This class provides the functionality to add a log entry to the Chart II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

3.12.1.9 EventConsumer (Class)

This interface provides the methods which any EventConsumer object that would like to be managed in an EventConsumerGroup must implement.

interface

3.12.1.10 Identifiable (Class)

This interface will be implemented by all classes which are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

interface

3.12.1.11 ObjectRemovalListener (Class)

This interface is implemented by objects that wish to be notified of objects being removed from the system. This is typically used by objects that store a collection of other objects, such as a factory, to allow them to remove objects from their collection when the object is to be removed from the system.

interface

3.12.1.12 OpLogQueue (Class)

This class is a queue for messages that are to be put into the system's Operations Log. Messages added to the queue can be removed in FIFO order.

3.12.1.13 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to

determine if a reconnect should be attempted, thus reconnects can be throttled independently of a supplier's push rate.

3.12.1.14 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

interface

3.12.1.15 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

interface

3.12.1.16 ServiceApplicationProperties (Class)

This class provides methods that allow the DefaultServiceApplication to access the necessary properties from the java properties configuration file. It also provides a default properties file which can be retrieved by anyone holding a ServiceApplication interface reference. This gives each installed service module the opportunity to load default values before retrieving property values from the properties file.

3.12.1.17 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

3.12.1.18 FMS (Class)

This class represents the CHART II system's interface to the FMS SNMP manager. Most methods included in this class have an associated method in the FMS SNMP Manager DLL provided by the FMS Subsystem. The other methods in this class exist to provide easier interface to the DLL. As an example, this class contains a blankSign method that actually calls setMessage on the FMS Subsystem with the message set to blank and beacons off.

3.12.1.19 FunctionalRight (Class)

This class acts as an enumuration that lists the types of functional rights possible in the CHART2 system. It contains a static member for each possible functional right.

3.12.1.20 IdentifiableLookupTable (Class)

This class uses a hash table implementation to store Identifiable objects for fast lookups.

3.12.1.21 IdentifierGenerator (Class)

This class is used to create and manipulate identifiers that are to be used in Identifiable objects.

3.12.1.22 QueueableCommand (Class)

A QueuableCommand is an abstract class used to represent a command that can be placed on a queue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed.

1

3.12.1.23 Log (Class)

Singleton log object to allow applications to easily create and utilize a LogFile object for system trace messages.

3.12.1.24 LogFile (Class)

This class creates a flat file for writing system trace log messages and purges them at user specified interval. The log files created by this class are used for system debugging and maintenance only and are not to be confused with the system operations log which is modeled by the OperationsLog class.

3.12.1.25 OpLogMessage (Class)

This class holds data for a message to be stored in the system's Operations Log.

3.12.1.26 PushEventConsumer (Class)

This class is a utility class that will be responsible for connecting a consumer implementation to an event channel, and maintaining that connection. When the verifyConnection method is called, this object will determine if the channel has been lost and will attempt to re-connect to the channel if it has.

3.12.1.27 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

3.12.1.28 TokenManipulator (Class)

This class contains all functionality required for user rights in the system. It is the only code in the system that knows how to create, modify and check a user's functional rights. It encapsulates the contents of an octet sequence that will be passed to every secure method. Secure methods should call the checkAccess method to validate the user. Client processes should use the check access method to verify access and optimize to reduce reduce the size of the sequence to only those rights which are necessary to invoke the secure method. The token contains the following information. Token version, Token ID, Token Time Stamp, Username, Op Center ID, Op Center IOR, functional rights

3.12.2 Sequence Diagrams

3.12.2.1 DefaultServiceApplication:shutdown (Sequence Diagram)

When the DefaultServiceApplication is shutdown, each of the ServiceApplicationModule object it created is shutdown. For those modules whose shutdown was successful, the offers they made are removed from the file that recorded the offers during the start of service. The connection to the database is cleared and the database object is deleted. The ServiceApplicationModule objects are deleted.

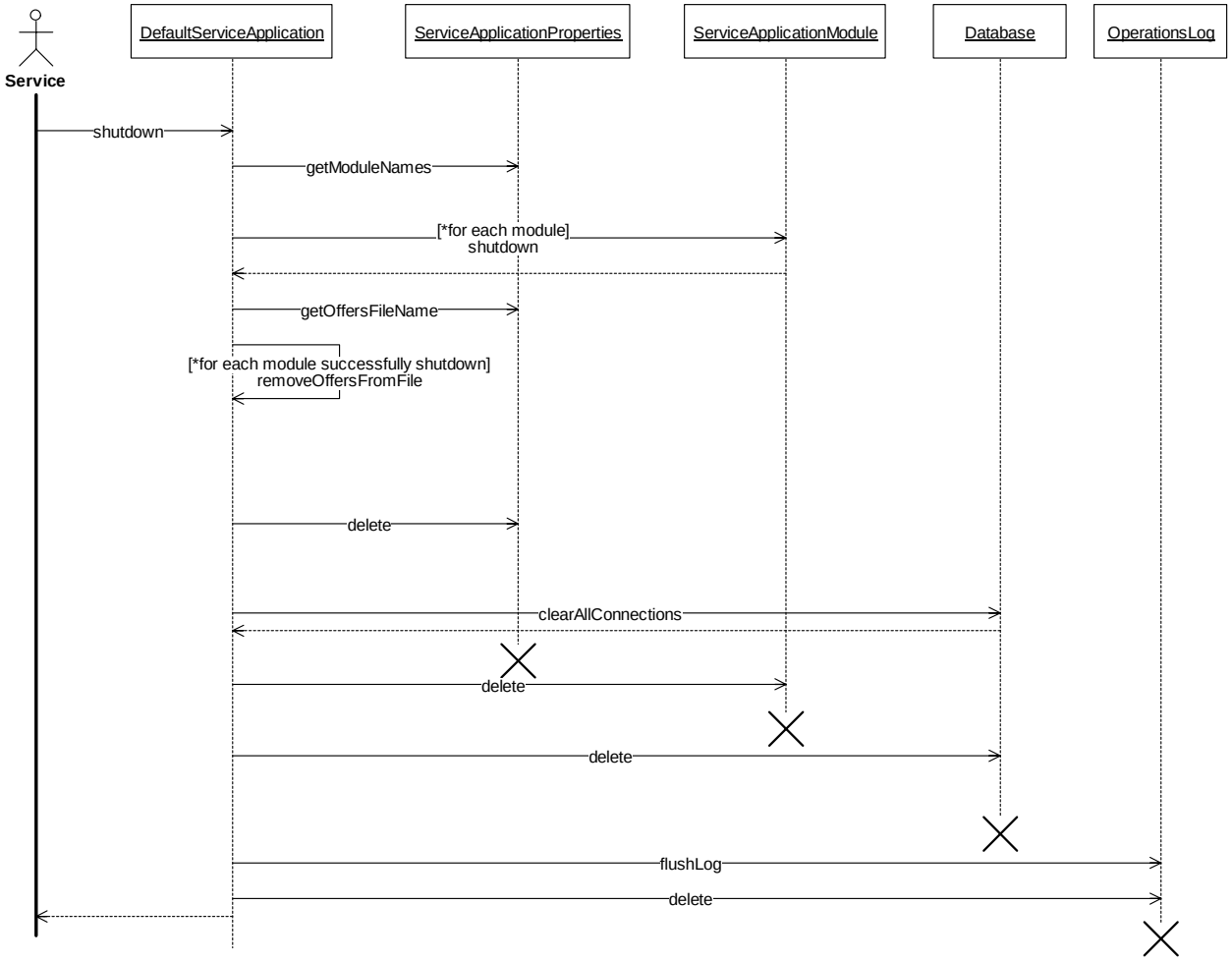

Figure 3-95. DefaultServiceApplication:shutdown (Sequence Diagram)

3.12.2.2 DefaultServiceApplication:Start (Sequence Diagram)

When a CHART2 service starts the DefaultServiceApplication, the ServiceApplicationProperties object, that encapsulates the operational parameters of the Chart2 system, is created. The CORBA objects ORB and BOA are initialized and their concurrency model and thread pool is configured. The Trader and Event Channel factory are acquired and the database object is created. During the start of a service, all the offers made by the service modules are recorded in a file (as will be seen later) and at the time of shutdown these offers are removed from the file. The presence of the offers in the file during start of service would indicate an improper previous shutdown. These lingering offers in the trader from the previous run of this service are withdrawn. The Service Application modules to be started by the service are determined from the ServiceApplicationProperties and the corresponding module class objects are instantiated. The modules are then initialized and the offers they made to the trader are recorded in a file.

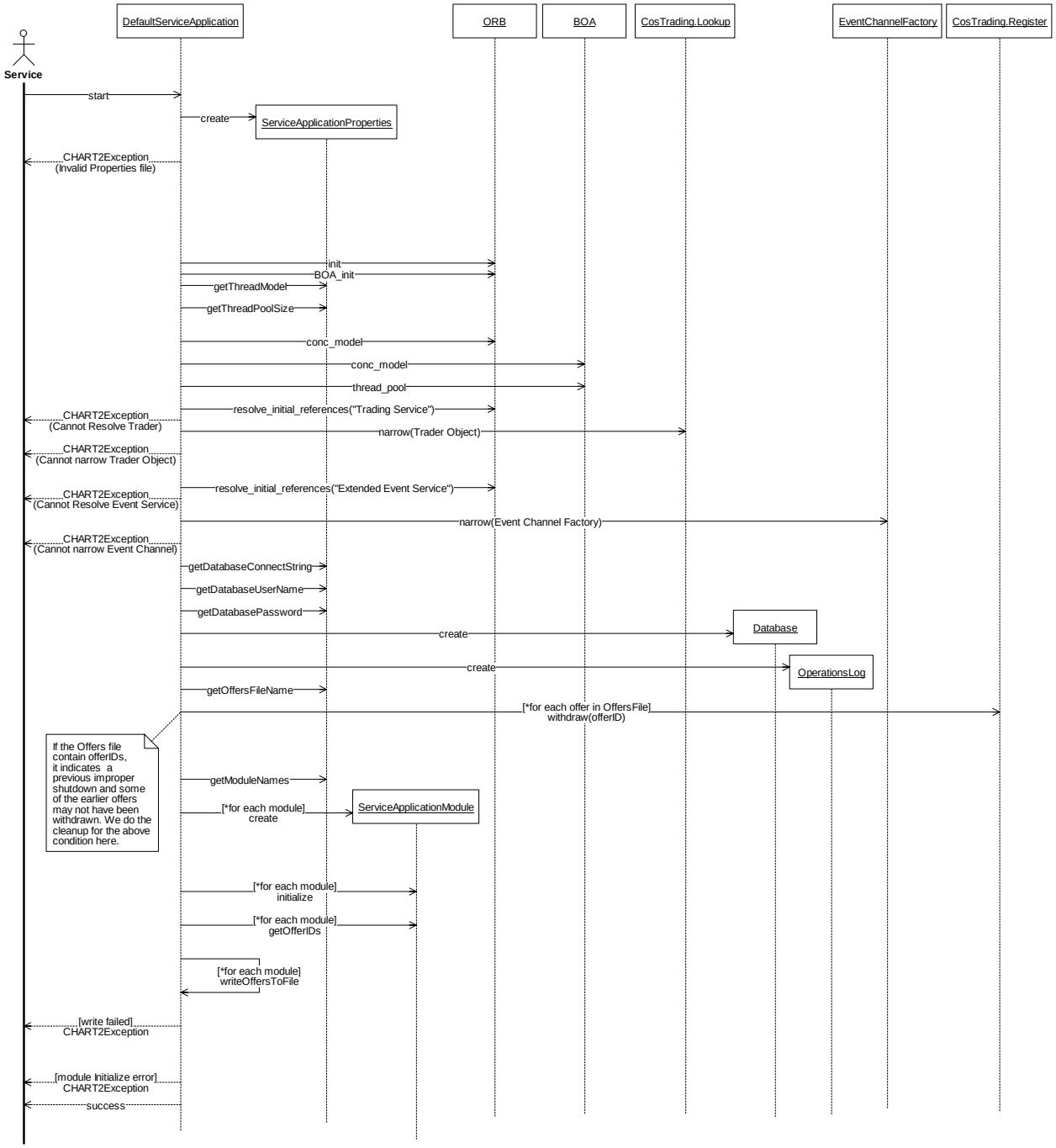

Figure 3-96. DefaultServiceApplication:Start (Sequence Diagram)

3.12.2.3 OperationsLog:LogMessage (Sequence Diagram)

When a log operation is invoked on the OperationsLog object, it creates a OpMessageLog and adds this object to the OpLogQueue. The OpLogQueue driver thread wakes up at a pre-configured interval and writes all the queued messages to the database.

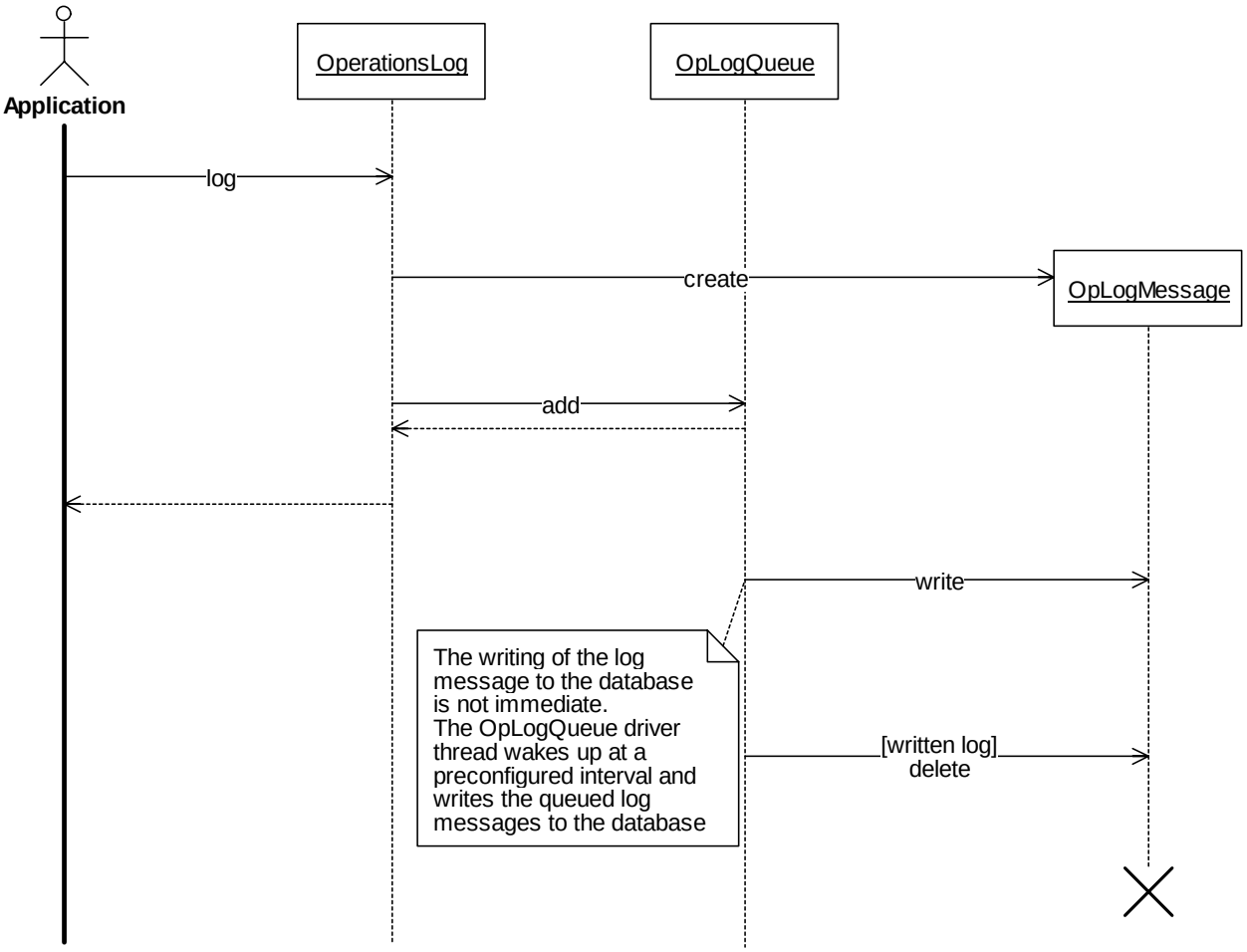

Figure 3-97. OperationsLog:LogMessage (Sequence Diagram)

3.13 CORBA Utilities

3.13.1 CORBAClasses (Class Diagram)

The CORBAUtilities package exists to provide reference to classes that are supplied by the ORB Vendor and are referenced by other packages' class or sequence diagrams.

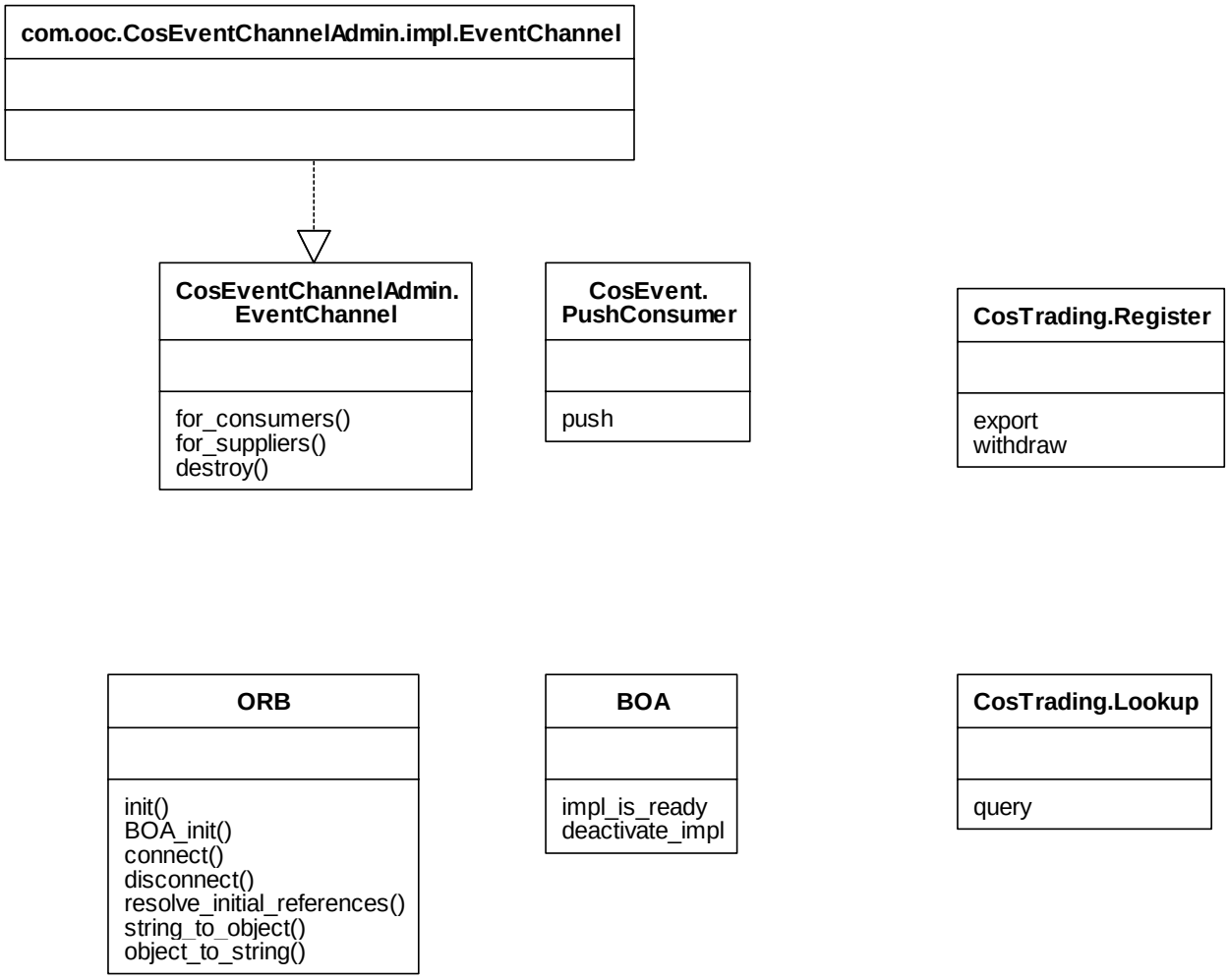

Figure 3-98. CORBAClasses (Class Diagram)

3.13.1.1 BOA (Class)

The BOA (Basic Object Adapter) is a class that assists implementation objects in using the ORB. Typical services provided include attaching and detaching object implementations to and from the ORB and generation of object references.

interface

3.13.1.2 com.ooc.CosEventChannelAdmin.impl.EventChannel (Class)

This class is the ORB vendor's implementation of a CORBA event channel. The event service provided by the vendor simply serves one of these objects. The Extended Event Service serves a factory that allows multiple instances of the vendor supplied event channel to be created.

3.13.1.3 CosEventChannelAdmin. EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

interface

3.13.1.4 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

interface

3.13.1.5 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects which have previously been published.

interface

3.13.1.6 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

interface

3.13.1.7 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

interface

3.14 Java Classes

3.14.1 JavaClasses (Class Diagram)

This package is included for reference to classes included in the Java programming language that are used in class and sequence diagrams for other packages within this design.

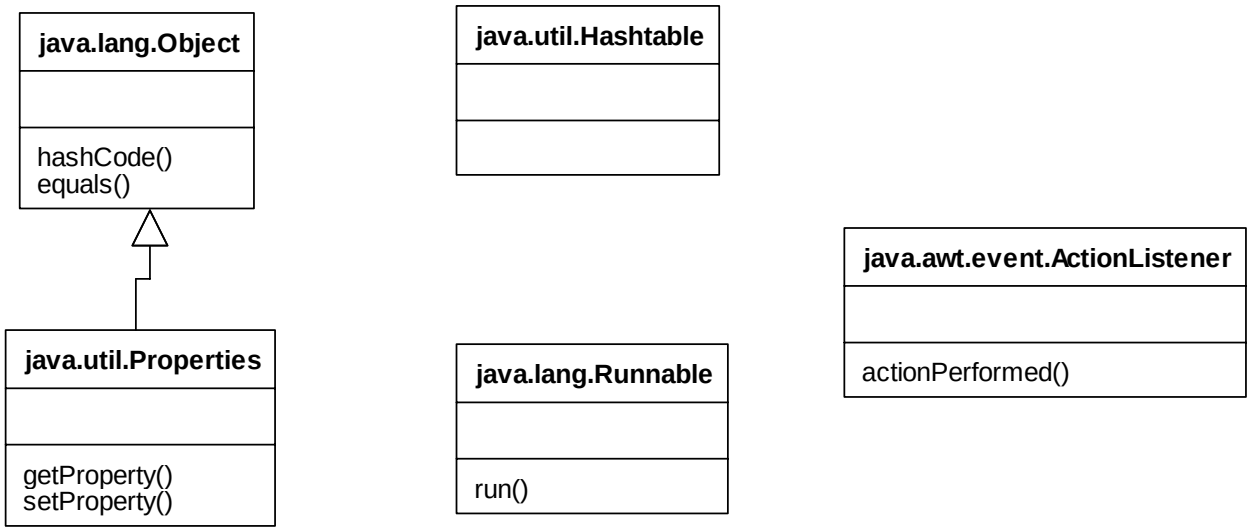

Figure 3-99. JavaClasses (Class Diagram)

3.14.1.1 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item is clicked. For menu items, it is attached to menu items when the menu is built.

interface

3.14.1.2 java.lang.Object (Class)

This is the base class from which all Java classes inherit.

3.14.1.3 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

interface

3.14.1.4 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method which is inherited by all objects from the java.lang.Object class.

3.14.1.5 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults;” this second property list is searched if the property key is not found in the original property list.

Appendix A - Glossary

Dictionary	A collection of banned words that cannot be used in a message that is displayed on a DMS or stored in a DMS message library.
DMS	A Dynamic Message Sign which can be controlled by one Operations Center at a time.
DMS Stored Message Item	A plan item that is used to set a specific message on a specific DMS when activated.
FMS	Field Management Station through which the CHART II system communicates with the devices in the field.
Functional Right	A privilege that gives a user the right to perform a particular system action or related group of actions. A functional right may be limited to pertain only to those shared resources owned by a particular organization or can pertain to the shared resources of all organizations.
Message Library	A collection of stored messages that can be displayed on the DMS.
Operations Center	A center where one or more users may log in to operate the Chart II system. Operations centers are assigned responsibility for shared resources that are controlled by users who are logged in at that operations center.
Organization	An organization is an agency that participates in the CHART II system and owns one or more Shared Resources.
Plan	A collection of plan items that can be activated as a group.
Plan Item	An action in the system that can be set up in advance to be activated one or more times in the future. Plan items must be contained in a plan. Specific types of plan items exist for specific functionality. A plan item carries out its specific task when activated.
Role	A Role is a collection of functional rights that a user may perform. The roles that pertain to a particular user for a particular login session are determined when he/she logs into the system.

Shared Resource	A resource that is owned by an organization. A user may be granted access to a shared resource owned by an organization through the functional rights scheme.
User	A user is somebody who uses the CHART II system. A user can perform different operations in the system depending upon the roles they have been granted in the system.